

LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84

Il6r
no. 776-781
cop. 2



The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

LIB 5 DEC 11

L161—O-1096

DESCRIPTION OF AN EXPERIMENTAL ON-LINE,
MINICOMPUTER-BASED INFORMATION RETRIEVAL SYSTEM

by

John Keith Morgan

February 1976



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN • URBANA, ILLINOIS

THE LIBRARY OF THE

1976 FEB 19 6

UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN



Digitized by the Internet Archive
in 2013

<http://archive.org/details/descriptionofexp779morg>

Report No. UIUCDCS-R-76-779

DESCRIPTION OF AN EXPERIMENTAL ON-LINE,
MINICOMPUTER-BASED INFORMATION RETRIEVAL SYSTEM

by

John Keith Morgan

February 1976

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

* This work was supported in part by the National Science Foundation under Grant No. US NSF DCR73-07980 A02 and was submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, February 1976.

ACKNOWLEDGMENT

Many people have contributed to the success of this project and the preparation of this thesis, and their help is gratefully acknowledged. Special thanks are due to my advisors, Professor David J. Kuck and Professor Duncan H. Lawrie for their advice, guidance, and support. Thanks, too, to the rest of the EUREKA gang: Dick Rinewalt, Mike Milner, and Bernie Hurley. Most of the system described in this thesis is the product of their labors, and apology is hereby tendered for any mistakes made in describing their portions of EUREKA.

Thanks also are in order for the continued moral support of my father and the typing and proofreading of Karen Hassett.

Finally, the financial support of the Department of Computer Science and the Veterans Administration has been invaluable.

TABLE OF CONTENTS

	Page
1. INTRODUCTION.....	1
2. USER'S GUIDE.....	4
2.1 INTRODUCTION.....	4
2.2 WHAT DOES EUREKA DO?.....	5
2.3 HOW DO I USE EUREKA?.....	8
2.4 PRELIMINARIES TO USING EUREKA.....	11
2.5 SEARCH SETS, QUERY SETS, AND QUERY NUMBERS - AN EXAMPLE.....	14
2.6 TERMINAL OPERATION.....	15
2.7 THE QUERY LANGUAGE.....	18
2.7.1 THE FIND STATEMENT.....	22
2.7.1.1 OPTIONS.....	25
2.7.1.1.1 CONTEXT CLAUSE.....	26
2.7.1.1.2 FROM CLAUSE.....	28
2.7.1.1.3 QUERY SET NAMING CLAUSE.....	31
2.7.1.1.4 COMMENTS CLAUSE.....	32
2.7.2 MACROS AND THE DEFINE STATEMENT.....	33
2.7.3 MAKE STATEMENT.....	35
2.7.4 CHANGE STATEMENT.....	37
2.7.5 COMMENT STATEMENT.....	38
2.7.6 DELETE STATEMENT.....	39
2.7.7 LOGON STATEMENT.....	42
2.7.8 LOGOFF STATEMENT.....	43
2.7.9 PRINT STATEMENT.....	43
3. SYSTEM PROGRAMMERS GUIDE.....	48
3.1 AN OVERVIEW OF EUREKA.....	48
3.2 ROOT TASK.....	55
3.3 USER INTERFACE.....	57
3.4 COMMAND PARSERS.....	58
3.5 SET HANDLER.....	59
3.5.1 USER FILE STRUCTURE.....	60
3.5.2 SET HANDLER TABLE.....	69
3.5.3 SET HANDLER SUPERVISOR.....	75
3.5.4 FILE WRITER.....	77
3.5.5 FILE READER.....	83
3.5.6 DELETE ROUTINE.....	86

3.5.7 RENAME ROUTINE.....	89
3.5.8 BITMAP HANDLER.....	89
3.6 SEARCH SUPERVISOR.....	93
3.6.1 SEARCH SUPERVISOR TABLE.....	94
3.6.2 SEARCH SUPERVISOR OPERATION.....	99
3.7 SET EXPRESSION EVALUATOR.....	102
3.8 INDEX AND POSTINGS HANDLER.....	103
3.8.1 FILES MANIPULATED BY THE INDEX AND POSTINGS HANDLER...	103
3.8.1.1 HASH1 TABLE.....	104
3.8.1.2 HASH2 TABLE.....	105
3.8.1.3 INDEX FILE.....	106
3.8.1.4 POSTINGS FILE.....	107
3.8.2 OPERATION OF THE INDEX AND POSTINGS HANDLER.....	108
3.9 MERGER.....	111
3.10 FULL-TEXT SEARCHER.....	114
3.10.1 FULL-TEXT SEARCHING ROUTINE.....	114
3.10.2 TEXT PRINTER ROUTINE.....	115
3.10.3 BROWSE MODE HANDLER.....	117
3.11 SET INFORMATION PRINTER.....	118

APPENDIX

A - Descriptions of Context Terms.....	122
B - Error Messages.....	124

REFERENCES.....	129
-----------------	-----

LIST OF FIGURES

	Page
3.1.1 SYSTEM FLOW DIAGRAM.....	49
3.2.1 USER LOGON BLOCK.....	56
3.5.1.1 USER FILE STRUCTURE.....	61
3.5.1.2 BLOCK 0 DETAIL.....	65
3.5.1.3 BLOCK 1-6 DETAIL.....	66
3.5.1.4 COMMENT DETAIL.....	67
3.5.2.1 READ/WRITE FORMAT (cmds 0-6,14,15).....	71
3.5.2.2 DELETE FORMAT (cmds 7-12,16,17).....	71
3.5.2.3 RENAME FORMAT (cmds 13,20).....	72
3.5.2.4 SET HANDLER COMMAND CODES.....	73
3.5.3.1 SET HANDLER SYSTEM FLOW DIAGRAM.....	76
3.6.1.1 SEARCH SUPERVISOR TABLE.....	95
3.6.1.2 TERM/SET QUAD DETAIL.....	96

INTRODUCTION

This thesis describes an experimental information retrieval system, EUREKA, designed and implemented at the University of Illinois by a research group under the direction of Dr. D.J. Kuck.

The EUREKA system was designed to provide a test system for studying several interesting problems in information retrieval, file manipulation, and large database systems. Whereas most current information retrieval systems are based on the use of predefined index terms for the retrieval of abstracts or titles of documents, EUREKA is organized around a database containing the entire text of documents, with each document indexed under every word occurring in the document (often referred to as inverted file organization). Beyond this assumption of file organization and content, the structure of EUREKA has been kept very modular in order to facilitate generation of variant systems for comparison of various methods of handling this type of data.

Some of the topics currently being studied include:

- 1) The effect of query language features on user performance;
- 2) Analysis of bottlenecks in information flow within the system;
- 3) Comparison of various levels of indexing, i.e. whether the inverted file should contain postings lists for documents or

for paragraphs within documents;

- 4) Effects of tradeoffs between the use of indexing to various levels and the use of full-text searching;
- 5) Design and analysis of special purpose hardware for use in information retrieval systems of this type;
- 6) Methods for handling non-textual information, such as tables and graphic items;
- 7) Development of automatic user aids to augment and improve the user's performance and recall/precision ratio;
- 8) Analysis of user performance on various types of retrieval problems using a variety of combinations of the aforementioned modifications of the basic EUREKA system to weigh the benefits of these modifications under different types of retrieval demands;
- 9) Collection of data for use in simulation studies of hardware/software systems for performing the data manipulation operations inherent in this type of system.

One can easily see from the above list of current research problems that the basic EUREKA system must be modified by various researchers in the process of their studies. This thesis is intended to provide the basic information necessary for performing these modifications. Chapter Two consists of the User's Guide to EUREKA, presenting the end user's view of the system currently in

use for studies of user performance. Chapter Three is the system documentation for the EUREKA system. This system documentation includes descriptions of most of the EUREKA information structures and the routines that manipulate them.

USER'S GUIDE

2.1 INTRODUCTION

EUREKA is an experimental information retrieval system based on full-text searching being constructed by a research group under the direction of Dr. David Kuck at the University of Illinois.

Since EUREKA is experimental in nature, it has been developed with ease of implementation, measurement, and modification occasionally taking precedence over ease of use.

One of the primary design goals of this project is to determine what features are necessary and/or desirable from a user's viewpoint. Our current query language is an attempt to provide a basic set of tools to the user in order that he/she may begin using the system and we may begin the process of monitoring and improving both the query language and the system. Hopefully, this will eventually lead to a better understanding of the man-machine interface problem, and hence, a better query language and system.

As one of the primary functions of EUREKA is to provide information about system use of hardware/software resources and user use of the system, all users and all processes will be monitored extensively. From this information and from user interviews and suggestions we hope to obtain a fairly clear view of what users expect from an information retrieval system and how best to provide for their needs.

Access to the system will be virtually unlimited, at least initially. User codes will be assigned so that each user may maintain private files on disk between sessions.

This manual is intended to serve as a guide for the inexperienced user and is therefore more verbose (and hopefully more helpful) than the usual user's guide to a system. Users with experience on other time-sharing or information retrieval systems may wish to merely skim the bulk of this guide, studying the examples and command definitions without spending too much time on the explanations.

2.2 WHAT DOES EUREKA DO?

EUREKA is a tool for use by anyone desiring to find specific information from a set of documents. More specifically, it allows a person desiring to find documents, authors, or even sentences within a document, that will

satisfy some set of restrictions specified by the user. A session at a terminal using EUREKA is equivalent to using a card catalog to find documents which might be of interest, then finding the documents in the stacks, selectively scanning through the documents to determine actual usefulness and find more possible reference terms under which useful information is likely to be found, and repeating this process for the new search terms until the required information is found. While this process might consume one or more days of the user's time if he/she were to conduct the search in person in a library, by using EUREKA he/she might well be able to accomplish his/her goals in a matter of minutes or an hour or two at most.

EUREKA accomplishes this by doing most of the searching, retrieving, and record keeping, allowing the user to concentrate on the intellectual aspects of the search. EUREKA allows the user to enter a search request consisting of a group of words the user feels characterize the information for which he/she is searching. This is equivalent to the user searching the card catalog for entries under those terms. Since EUREKA is a full-text searching system, every document is indexed under every word contained within the text of that document, rather than under only a few index terms as in a card catalog. This allows the user much more freedom in

selecting search terms and in forming his/her search strategy. Another feature of EUREKA not feasible in a card catalog is allowing the user to specify the context within which the search terms must occur. In EUREKA a user may specify that several words must occur within the same sentence or paragraph, etc., rather than merely occurring anywhere in the document. EUREKA thus frees the user from many of the trivialities of searching through the card catalog trays, worrying about alphabetical sequence, trying to find terms in a strictly controlled index vocabulary that adequately describe his/her request, and locating the document in the stacks. It allows the user to see the results of his/her search strategy immediately, without having to leave the place where he/she conducts his/her search to locate the documents before determining its relevance. Once the document has been retrieved by EUREKA, the user may use EUREKA to view any portion of the document on-line and selectively print any portion of the document (up to and including the entire document) on-line. This also allows the user to find new search terms and evaluate and modify his/her search strategy accordingly without a tedious and time-consuming search session in the stacks.

Another useful feature of EUREKA is the record keeping function it performs for the user. A user is allowed to keep a record of all of the searches he/she has already completed and the results of these searches. The user may also attach comments to these documents and search results that he/she may view later on-line to assist in keeping track of what he/she has already done. Various other aids exist for assisting the user in fulfilling his/her information requirements with a minimum of effort.

2.3 HOW DO I USE EUREKA?

Using EUREKA is relatively simple. EUREKA functions primarily by determining whether or not certain words appear in a document. Thus, by using a very simple set of commands, the user may direct the system to find all documents containing some combination of words he/she feels will characterize documents in which he/she is interested. By specifying various options (as will be explained in a later section) the user may view selected sentences, paragraphs, titles, etc. from the documents retrieved by EUREKA. Whether or not any options are selected, the system will respond with a list of all documents containing the words or phrases specified by the user and the relative rank of each document. This relative rank is computed for each document from the

number of occurrences of each search term in the document. The document containing the largest total number of occurrences of search terms is ranked number one, the next largest number two,..., down to the document containing the least total number of occurrences of all the search terms. At this point the user may use other commands to view portions of the documents just retrieved on-line. By doing so he/she may evaluate the actual relevance of each document, and may also find new search terms to use in searching for more documents.

The list of all documents retrieved by a query statement is saved by the system for later use by the user. After conducting several such searches, thus generating several document lists (called query sets), the user may use other commands in the query language to compare and combine the query sets to generate new lists of documents that meet his/her requirements more exactly.

For example, consider a legal secretary searching for legal statutes pertaining to roads near cheese factories, but not pertaining to interstate highways. If this user were to use EUREKA to search the State Statutes data base, a possible search strategy would be:

- 1) Search for all documents (actually chapters of the State Statutes) containing the words "CHEESE" and

"FACTORY" in the same sentence.

2) Search for all documents containing the word "ROADS".

3) Search for all documents containing both the words "INTERSTATE" and "HIGHWAYS" within the same sentence.

At this point the searcher has three lists of documents (query sets) that he/she may compare and combine:

4) Compare the list of documents responding to query #1 to the set of documents responding to query #2, selecting the documents that appear in both lists. This gives us a set of documents pertaining to both cheese factories and roads.

5) Compare the list of documents generated in step 4 (query set #4) to query set #3, selecting only those documents appearing in query set #4 but not in query set #3. This eliminates any documents referring to interstate highways.

The user may then use other commands of the system to view portions of the documents (or have them printed on a line printer) and discover that cheese factories may not be located within four hundred feet of a dirt road. If satisfied the user may log off, or if not, continue his/her search using new search terms.

The EUREKA commands to perform all of the preceeding actions would be :

LOGON ANYBDY

FIND 'CHEESE' * 'FACTORY' IN SENTENCE = CHEZFAC

FIND 'ROADS' FROM ALL = ROADLIST

FIND 'INTERSTATE' * 'HIGHWAYS' FROM ALL = TURNPIKES

MAKE CHEZFAC * ROADLIST = TEMPLIST

MAKE TEMPLIST - TURNPIKES = FINALLIST

All of the above commands will be explained in detail in a later section.

2.4 PRELIMINARIES TO USING EUREKA

First let us define some terms:

DOCUMENT:

One logical division of text that is given a document number and is indexed by every word that occurs within the bounds of that division. Size and logical division may vary between data bases. In the information retrieval data base one journal article is taken to be one document, while in the State Statutes data base one chapter of the statutes is taken to be a document. In the business abstracts data base, each abstract is a document. When more data bases are added ad hoc divisions will be determined for them.

DOCUMENT NUMBER:

A number arbitrarily assigned to each document by EUREKA so that unnecessarily long document names do not have to be remembered and handled by the user or the system.

QUERY:

A command to EUREKA in the EUREKA query language. Usually a search command (FIND statement) or a query set manipulation command (MAKE statement) that generates a query set.

QUERY NUMBER:

Each query entered by the user is automatically assigned a number (which is printed out each time EUREKA notifies the user of its readiness to accept a new command) by which the user may refer to the results of that query or to any comments attached to that query by the user.

QUERY SET:

The list of documents retrieved by a query. This query set may be referred to by query number or by a user assigned query set name at a later time for use in other queries or PRINT statements.

SEARCH SETS:

It is assumed that the user of EUREKA will normally wish to conduct his/her searches in gradual steps rather than by one gigantic, complicated query statement. In most

cases this will involve conducting a search on fairly general search terms first and then narrowing down the number of documents retrieved by searching just those documents that responded to the first query on yet other search terms. To facilitate this mode of operation EUREKA allows the user to specify any query set (or combination thereof) as the set of documents to be searched by a new search statement (FIND statement). This set of documents from which the search is to be conducted is referred to as the search set. The search set is specified by the inclusion of an optional "from" clause on the query language statement that specifies the search to be performed (FIND statement). If no search set is specified by the user, "FROM ALL" is assumed. That is, we assume that the user wishes to search from the entire data base rather than any restricted set.

CONTEXT:

Each document is divided into several parts, such as title, author, body, abstract, sentence, etc. Each of these divisions is referred to as a context. Note that the contexts "sentence" and "paragraph" may occur arbitrarily many times in each document, but all other contexts occur only once per document.

2.5 SEARCH SETS, QUERY SETS, AND QUERY NUMBERS - AN EXAMPLE

Let us consider a person searching the information retrieval data base for information concerning the application of formal languages and automata theory to information retrieval. After logging into EUREKA (to be explained later), EUREKA would type out:

QUERY #00001:

#

This is the signal to the user that EUREKA is ready to accept query number one (notice that 1 is the query number). One possible search strategy is to first search on the terms "FORMAL" and "LANGUAGE", requesting a list of all documents that have one or more occurrences of both search terms in them. The list of documents that EUREKA responds with is called the query set. As soon as EUREKA completes typing out the list of documents containing both the words "FORMAL" and "LANGUAGE", it will advance to a new line and type:

QUERY #00002:

#

The user could then enter query #2 which would consist of a search statement asking for a list of documents containing the word "AUTOMATA". If the user specifies "FROM LAST" EUREKA will assume that he wants to search only those documents that are members of query set #1 (that is, documents that contain both the word "FORMAL" and the word "LANGUAGE"). The user

could alternatively specify that the entire data base was to be searched by not specifying any "from" clause. This second query statement would generate a new list of documents which would be query set #2. Both query set #1 and query set #2 can now be referred to by later "from" clauses in determining a search set.

2.6 TERMINAL OPERATION

The terminal currently in use with EUREKA is an Infoton CRT terminal. The keyboard is very similar to an electric typewriter. There are no lower case letters. Since the shift key no longer is used to supply capital letters instead of lower case, it is used on the terminal keyboard to supply special characters (such as ,<,>,"','and []). These special characters are, for the most part, printed above the letters on the terminal keyboard.

One other point about keys should be noted. On a terminal keyboard zero and the letter "O" are not the same. The zero key is between the "9" key and the ":" key on the top row of keys, while the "O" key is between the "I" and "P" keys on the next row down. Also note that there are both single quote and double quote keys, the double quote key being SHIFT-2 and the single quote key being SHIFT-7. EUREKA makes

use of both single and double quotes, so do not attempt to use two single quotes where one double quote is called for.

The only other special keys the user need be aware of are the CTRL key, the RETURN key, and the RUB OUT key. The RETURN key is used to cause a carriage return and to cause the line of characters you have just typed on the keyboard to be sent to EUREKA. While you type the terminal holds what you type in a temporary storage area until you push the RETURN key, at which time the entire line is sent to EUREKA. If, while you are typing in a command to EUREKA, you discover that you have just hit the wrong key, you may delete the last letter you typed in by hitting the RUB OUT key. Pushing the RUB OUT key twice deletes the last two characters, etc. The CTRL key is similar to the SHIFT key in that it assigns a new set of meanings to other keys on the keyboard. If, while typing in a line you notice a mistake back in the first part of the line and don't want to use the RUB OUT key to rub out the entire line back to that point, you may delete the entire line by holding down the CTRL key and pushing the "U" key.

The user should note that EUREKA can send and receive commands at the same time, even though commands typed in while EUREKA is working on something else are not echoed on the terminal until EUREKA finishes whatever it was working on.

Therefore, it is unwise to play with the keyboard while EUREKA is working on a command you have just entered since whatever you type in will eventually be sent to EUREKA.

In order to prevent information being flashed on the screen faster than you can read it, EUREKA pauses every 15 lines or at the end of each item you have requested to have printed, whichever occurs first. If the line count has been reached, EUREKA will print an "@" on the next line of the screen to inform you of this fact. If the end of an item has been reached, EUREKA uses an "!" to signal you. When signalled by either an "@" or an "!", you may then instruct EUREKA to continue with the current output by pushing the carriage return key, stop the current output (the current search continues, however) by pressing the "K" key followed by a carriage return, skip to the next document in the list to be printed by pressing "S" followed by a carriage return, or enter the browse mode to skip around within the document currently being printed.

Browse Mode

Browse mode is merely a method of viewing sentences or paragraphs adjacent to the one currently being printed. To skip back to the previous sentence, type in "-S" and push the

carriage return key. Similarly, the previous paragraph may be viewed by typing "-P". To skip forward one sentence or paragraph, type "+S" or "+P". If, while viewing portions of a document, you decide to look at the title or author (or any other context), you have only to type in the correct context identifier (see Appendix A) and push carriage return when EUREKA has paused to allow you to respond. Once any of the commands for skipping around within a document or printing other contexts have been used after a "@" or "!" flag, the user is said to be in browse mode. In order to get out of browse mode and resume the output from the current document, type in "E" followed by a carriage return.

2.7 THE QUERY LANGUAGE

There are only nine commands in the EUREKA query language. Only two of these are necessary for conducting searches, while the other seven perform auxiliary functions. In brief, the functions of these commands are:

FIND:

The FIND statement is the heart of the EUREKA system. It is used to perform searches for documents containing a user selected set of words or phrases.

MAKE:

The MAKE statement is used to compare and combine sets of documents created by the FIND statement.

COMMENT:

The COMMENT statement is used to write notes to yourself concerning a query set or particular document. These notes may be retrieved at a later time by use of the PRINT statement.

CHANGE:

The CHANGE statement is used to assign a name to a query set or to change the existing name of a query set.

DELETE:

The DELETE statement is used to delete query sets, macros, and/or comments which are no longer needed.

PRINT:

The PRINT statement is used to print user comments , selected portions of a document (up to and including the entire document), and information about preceeding queries and their resultant query sets.

LOGON:

The LOGON statement is used to identify the user to EUREKA in order for EUREKA to gain access to the correct user files and data base.

LOGOFF:

The LOGOFF command is used to terminate a session. It disconnects a user from the EUREKA system and closes his files.

DEFINE:

The DEFINE statement is used to give a name to a list of search terms so that the user does not have to repeatedly type in long search expressions. These macro definitions are saved in the user file area and may be used in conjunction with other search terms in FIND statements.

Each of the query language commands has a very simple basic form which may be used alone or with the addition of optional clauses that significantly increase their power. This allows the user to begin with very simple query statements and progress to more complicated forms when the need arises.

EUREKA is a keyword driven language. This means that EUREKA figures out what a command typed in by the user means by looking for special words that tell it to do a specific operation. These keywords are usually followed by one or more user supplied parameters that control how the operation is performed.

Although this sounds somewhat complicated, it is pretty simple. Algebra is one other example of a keyword driven language. In the algebraic expression:

$$A = 3X + 5$$

the equal sign "=" is a keyword to tell anyone looking at it that whatever appears on the right side of it is equal to whatever appears on the left. Similarly, the plus sign, "+", is a keyword for the add operation, while A, 3, X, and 5 are parameters.

In discussing the EUREKA query language we will wish to represent parameters in a general fashion in addition to giving specific examples, since it would take an impossibly large number of examples to cover the possible range of each command entirely. Therefore, if we wish to describe a parameter, we will just use an English phrase that describes the parameter and then describe allowable forms for the parameter by giving a definition for the phrase. However, this causes a slight problem. Since most of the EUREKA keywords are English words, we need something to distinguish the EUREKA keywords from the English phrases! For this purpose we will use some characters not used in the EUREKA query language to enclose the English phrases. Since neither the less-than symbol (<) nor the greater-than symbol (>) is used in the EUREKA language, we will use them for separating

the phrases from the keywords. For example, the algebraic expression used earlier could be represented by:

$$\langle \text{VARIABLE} \rangle = \langle \text{FORMULA} \rangle$$

where $\langle \text{FORMULA} \rangle$ is of the form:

$$\langle \text{DIGIT} \rangle \langle \text{VARIABLE} \rangle + \langle \text{DIGIT} \rangle$$

and so forth.

The notation just presented is definitely easier to understand after seeing several examples than it is to describe formally. The user should become familiar enough with the notation to make sense of it by comparing examples to the description in the above described notation for the next few pages.

2.7.1 THE FIND STATEMENT

The FIND statement is used to enter search requests. Its basic form is:

$$\text{FIND } \langle \text{SEARCH EXPRESSION} \rangle$$

The keyword is "FIND", but for brevity this may be abbreviated "F". The parameter is a combination of words enclosed in single quotes for which the search is to be conducted. Examples are:

$$\text{FIND 'PRECISION'}$$

$$\text{FIND 'PRECISION' * 'RECALL'}$$

P 'CATS' * 'DOGS' + 'MICE'

In the above examples, the first one directs EUREKA to find all documents in which the word "PRECISION" appears. The second example directs EUREKA to find all documents in which both the word "PRECISION" and the word "RECALL" appear. Note that "*" means that "Both what is on the right and what is on the left must appear". The final example directs EUREKA to find all documents in which both the word "CATS" and the word "DOGS" appear, and to also retrieve all documents in which the word "MICE" appears. Note that "+" means that "Either what is on the left or what is on the right must appear". For this reason, "+" is referred to as "OR", while "*" is referred to as "AND".

Another fact to note about the final example is that the "AND" is considered before the "OR". That is, the third example is taken to mean

"All documents containing either the word "MICE" or both the word "CATS" and the word "DOGS"

rather than

"Find all documents containing both the word "CATS" and either of the words "DOGS" and "MICE".

A little perusal will show the reader that the above two sentences do not mean the same thing. This is similar to the problem in algebra of deciding whether

$$A = 4/X-5$$

means

$$A = 4/(X-5)$$

or

$$A = (4/X) - 5$$

Since EUREKA always assumes "AND" is to be done before "OR", if one wishes to tell EUREKA to find "all the documents containing the word 'CATS' and either of the words 'DOGS' or 'MICE'", one must make use of parentheses to alter the order of evaluation of the search expression. For instance:

F 'CATS' * ('DOGS' + 'MICE')

One final remark about search expressions. The words in single quotes are called "search terms" and are used in the exact form typed in when searching for matches in the document texts. Therefore, if one types in

FIND 'DOG'

one gets the list of documents containing the word "DOG", but not those containing the words "DOGGIES", "DOGS", etc. (unless the word "DOG" appears in them also). If the user wishes to search for all forms of a word stem, then he may make use of the "universal character", "#". For instance, the query

F 'DOG#'

would return the list of all documents containing any of the

following words: "DOG", "DOGS", "DOGMATIC", "DOGWOOD", ...etc. This is known as suffixing, since it directs EUREKA to accept any suffix attached to the word stem. Prefixing is also permitted. For instance,

F '#FIX'

would retrieve all documents containing any of the following words: "PREFIX", "POSTFIX", "SUFFIX", ...etc. Both prefixing and suffixing may be performed at once. The query

F '#IZ#'

would retrieve all documents containing such words as: "AMERICANIZATION", "AMERICANIZED", "COMPUTERIZED", etc. However, if the universal character appears in the middle of a word it is assumed to actually be a pound sign. Therefore,

F 'A#D'

tries to find the word "A#D" rather than retrieving all documents containing a word starting with "A" and ending with "D".

2.7.1.1 OPTIONS

Now we may begin to add on optional clauses to the basic FIND statement in order to simplify certain operations and make possible others that are not possible with the basic FIND statement. Options are just like options on a car - they may be included if necessary or left out if not needed.

2.7.1.1.1 CONTEXT CLAUSE

The first optional clause we shall discuss is the context option. its general form is:

FIND <Search Expression> IN <Context>

"In" is the keyword to inform EUREKA that a search context follows, and <Context> is the parameter that specifies the context in which the words in the <Search Expression> must appear in order for the document to be retrieved. For example, the query:

FIND 'DOG' * 'CAT' IN SENTENCE

directs EUREKA to retrieve all documents in which both the word "CAT" and the word "DOG" appear in the same sentence within the document.

The list of all allowable contexts is:

1 : SENTENCE	2 : PARAGRAPH	3 : DOCUMENT
4 : ARTICLE	5 : DATA	6 : AUTHOR
7 : TITLE	8 : SOURCE	9 : DATE
10: PAGES	11: MISC	12: INDEX
13: KEYS	14: TEXT	15: ABSTRACT
16: BODY	17: NOTES	18: REFERENCES
19: COMMENTS		

Definitions of the various contexts appear in Appendix A. Note that any context term may be abbreviated by truncating it to any length that leaves it distinguishable from all context terms preceeding it in the above list. For instance, specifying "IN A" for a context is the same as specifying "IN ARTICLE", while "IN AU" specifies "IN AUTHOR". Examples of FIND statements containing context clauses are:

F 'SALTON' + 'LANCASTER' IN AUTHOR

Which directs EUREKA to find all documents written by either Salton or Lancaster.

FIND 'GARBAGE' IN COMMENTS

Which directs EUREKA to find all documents to which the user has added a comment (to be explained later) containing the word "GARBAGE"

F 'COMPUTER#' * 'LIBR#' IN TITLE

which directs EUREKA to find all documents that contain both a word starting with the characters "COMPUTER" and a word starting with the letters "LIBR" in the title.

F 'AUTOMATA' IN AB

which directs EUREKA to find all documents containing the word "AUTOMATA" in their abstract.

2.7.1.1.2 FROM CLAUSE

The next option we shall discuss is the from clause. the from clause is used to specify the search set (set of documents among which the search is to be conducted. See Section 2.4). Its general form is:

FIND <Search Expr> FROM <From Set>

The keyword is "FROM", which directs EUREKA to search for the words in the <Search Expression> only among the documents that meet the requirements of the set expression <From Set>. The parameter <From Set> is an expression involving query sets and documents. Query sets may be referred to by either query number or query set name, while documents are referred to by a list of document numbers separated by commas, enclosed by square brackets. The general form for a <From Set> is:

<Set Term> <Set Op> <Set Term> <Set Op>....<Set Term>

where <Set Term> is either a query set number, a query set name, or a document list as described above. <Set Op> is one of the following:

"*", "+", or "-".

Since the concept of a <From Clause> is difficult to describe rigorously in English, let us resort to some examples.

FIND 'ALPHA' FROM 1

directs EUREKA to find all documents that responded to query #1 and also contain the word "ALPHA".

F 'ALPHA' FROM 1 * 2

directs EUREKA to find all documents that responded to both query #1 and query #2 and also contain the word "ALPHA".

F 'ALPHA' FROM 1 * 3 + 2

directs EUREKA to find all documents that responded to either query #2 or to both query #1 and query #3, and that in addition, contain the word "ALPHA".

FIND 'ALPHA' FROM 1 - 2

directs EUREKA to find all documents that responded to query #1 but did not respond to query #2 , and also contain the word "ALPHA".

F 'ALPHA'*'SOMETHING' IN SENTENCE FROM 1+[1,24,3]

directs EUREKA to find all documents that responded to query #1 and contain the words "ALPHA" and "SOMETHING" in the same sentence, and to also search documents 1, 24, and 3 for the occurrences of the search terms.

FIND 'ALPHA' + 'BETA' FROM 1 -[3,19]

which directs EUREKA to search all documents responding to query #1 except documents #3 and #19 for an occurrence of either the word "ALPHA" or the word "BETA".

F 'ALPHA'

directs EUREKA to search all documents in the data base for any that contain the word "ALPHA".

F 'ALPHA' FROM LAST

EUREKA is directed to search for documents containing the word "ALPHA" among all documents responding to the last query. I.E., if this is query #4 then all documents responding to query #3 are used as the search set (just as if "FROM 3" had been specified). However, if this happens to be query #1, then the entire data base is searched because there are no preceeding queries from which to search.

F 'ALPHA' FROM CHEZFAC - 3

This directs EUREKA to search for documents containing the word "ALPHA" among all the documents in the query set named "CHEZFAC" by the user, except any documents that responded to both query #3 and the query named "CHEZFAC" by the user. Let us note in passing that "*" is used as a Boolean "AND" operator, "+" is used as a Boolean "OR" operator, and "-" is the Boolean "RELATIVE COMPLEMENT". Note also that parentheses may not be used to alter the order of evaluation of the set expression. If one wishes to have a complicated expression of set name/numbers not obtainable by the from clause, one must use the MAKE statement (see Section 2.7.3) to obtain a set equivalent to the desired set expression.

2.7.1.1.3 QUERY SET NAMING CLAUSE

Since most of us will not want to keep track of large numbers of relatively easy to forget query numbers, EUREKA allows the user to specify a mnemonic name for any query set he/she creates. One method of assigning a set name is via the set name clause attached to either a "FIND" or "MAKE" statement (another method is via the "CHANGE" statement, which will be described later).

The general form for the query set name clause is:

FIND <Search Expression> = <Setname>

in which "=" is the keyword that signals EUREKA that what follows is a name the user wishes to have associated with the query set that will result from this FIND statement. <Setname> may be any string of up to ten letters and/or numbers (no special characters like +, ", or <) that meets the following restrictions:

- 1) Must not begin with a number
- 2) Must not be any of the following words:

"ALL", "FROM", "COMMENTS", "MACRO", or "LAST".

Examples of find statements containing set naming clauses are:

FIND 'DOG#' FROM 3 = DOGSET

which directs EUREKA to search all documents in query set #3 for words that begin with the letters "DOG" and then name the

resulting query set "DOGSET".

F 'ANORAK' + 'CAGOULE' IN TITLE FROM ALL = RAINCOATS
which directs EUREKA to search the entire data base for documents containing either the word "ANORAK" or the word "CAGOULE" and then name the resulting query set "RAINCOATS".

F 'CHEESE' * 'FACTOR#' IN SENTENCE = CHEZFAC

Now turn back to Section 2.3 and study the example there.

2.7.1.1.4 COMMENTS CLAUSE

The next option we shall discuss is the comments clause, which is used for attaching user comments to a query set. Comments are a mechanism for writing notes to oneself that may be retrieved at a later time via the PRINT statement. These comments may be a statement of the purpose of creating this particular query set, the number of documents in the set, or anything else the user feels to be of interest.

The general form for the comments clause of the FIND statement is:

FIND <Search Expr> "<Comment String>"

In this case the keyword is in two parts, the two double quotes. The comment string itself can be any string of words, numbers, etc. up to 256 characters in length. The only restriction on the character string is that, since EUREKA uses

double quotes as flags for finding the beginning and end of the comment, if the user wishes to have double quotes appear in his comment string he must enter two double quotes side by side.

Examples of FIND statements with comment clauses are:

```
FIND 'CATS' * 'DOG#' = ANIMALSET "FIND CATS & DOGS"
```

```
FIND 'DOG#' "TRY OUT ""UNIVERSAL CHARACTER"""
```

Note that if the user requests (via the PRINT statement) to have the comments from the second example printed, it will appear as:

```
TRY OUT "UNIVERSAL CHARACTER"
```

2.7.2 MACROS AND THE DEFINE STATEMENT

We must now explain the use of macros so that references to them will be only confusing rather than incomprehensible in the descriptions of other commands.

In order to avoid forcing the user to type in long search term expressions repeatedly when conducting several searches, each of which contains some of the same sub-expressions, we allow the user to define sub-expressions of search terms as macros. An example of a find statement using a macro is:

```
FIND 'CATS' * BUGS
```

where "BUGS" has previously been declared (by a DEFINE

statement) to be 'TICKS'+'FLEAS'. This FIND statement is equivalent to:

```
FIND 'CATS' * ('TICKS' + 'FLEAS')
```

Note that when macros are used in FIND search expressions, they are not delimited by single quotes as are search terms. This is to help EUREKA determine whether a term is actually a search term or a macro that must be expanded. Note also that a macro text is enclosed by parentheses, thereby possibly altering the order of evaluation of the term expression.

Macros are declared by the use of a DEFINE statement, which has the following format:

```
DEFINE <Search Expression> = <Macro Name>
```

In the DEFINE statement, "DEFINE" is the keyword that tells EUREKA what to do with the rest of the command, <Search Expression> is as defined for the "MAKE statement" (Section 2.7.3). The "=" is a keyword to separate the search expression from the name the user wishes to assign to the macro (<Macro Name>). The <Macro Name> must follow the same rules set forth for the <Set Name> (Section 2.7.1).

Examples of DEFINE statements are:

```
DEFINE 'TICKS' + 'FLEAS' = BUGS
```

which defines the macro used in the macro example above.

```
DEF 'FLEAPOWDER' * BUGS = CURES
```

Note that macros may be used within definitions of new macros. Also note that the keyword "DEFINE" may be abbreviated "DEF".

2.7.3 MAKE STATEMENT

The MAKE statement is used to compare two or more query sets and generate a new query set based on the results of the comparison. The basic form of the MAKE statement is:

```
MAKE <Set Expr>
```

where "MAKE" is the keyword (which may be abbreviated "M"), and <Set Expr> is the same as the <From Set> defined for the FIND statement (see Section 2.7.1). This is because the MAKE statement is, in effect, an explicit method of creating new sets of documents from old sets and explicit document numbers. The difference between <From Sets> created by a from clause of a FIND statement and a query set created by a MAKE statement is that a <From Set> is temporary only and may not be referred to again without explicitly re-creating it, while a query set created by a MAKE statement is given exactly the same status as a query set created by a FIND statement. It may be named, have comments attached to it, and it may be referred to by query number or query name in a later query.

Examples of basic MAKE statements are:

```
MAKE 3 * 2
```

which directs EUREKA to create a new query set consisting of all documents that appear in both query set #3 and in query set #2.

```
MAKE 3 + 2 * CHEZFAC
```

which directs EUREKA to create a new query set composed of all documents that are either in query set #3 or in both query set #2 and the query set named "CHEZFAC" by the user.

```
MAKE 3 + [7,26,8]
```

which directs EUREKA to create a new query set composed of documents 7, 26, and 8, and also all documents that are in query set #3.

The options for the MAKE statement are the query naming clause and the comments clause. Both of these options function exactly like their counterparts for the Find statement, so the reader is directed to Section 2.7.1 if further description is required. The general form of the MAKE statement is:

```
MAKE <Set Expression> = <Setname> "<Comment String>"
```

Examples of MAKE statements are:

```
MAKE 3 + FINALSET = NEWSET "CREATE NEW SET FROM  
FINALSET & 3"
```

```
M 3 * CHEZFAC + [7,13] "NO SET NAME"
```

M ROADSET - [11,7] = NEWROADSET "DELETE DOCS 11&7 FROM ROADSET"

2.7.4 CHANGE STATEMENT

The CHANGE statement is used to assign a name to a query set or to change an existing name of a query set. Its general form is:

CHANGE <Query Set ID> TO <Query Set Name>

"CHANGE" is the keyword, which may be abbreviated "CH", that informs EUREKA that a name assignment/change follows. "TO" is a keyword to separate the old set ID from the new. <Query Set ID> is either a query number or a query set name. <Query Set Name> is the new query set name to be assigned to the query set identified in <Query Set ID>. This name must obey the rules described for query set naming in Section 2.7.1. Examples of CHANGE statements are:

CHANGE 3 TO GOODSET

which assigns the name "GOODSET" to query set #3.

CH GOODSET TO BADSET

Which changes the name of the query set currently named "GOODSET" to "BADSET".

Macros may be renamed by following the word "CHANGE" by the word "MACRO". An example is:

```
CHANGE MACRO TX34J TO FRED
```

which changes the name of a macro named "TX34J" to "FRED". Note that the keyword "MACRO" may not be abbreviated.

2.7.5 COMMENT STATEMENT

The COMMENT statement is used to assign comments to query sets or individual documents. These comments may be retrieved upon demand by use of the PRINT statement. The general form of the COMMENT statement is:

```
COMMENT <Set/Document ID> "COMMENTS"
```

"COMMENT" is the keyword (which may be abbreviated "CO") to inform EUREKA that what follows is a set or document identifier for the set the user wishes to add a comment to, and the double quotes act as delimiters for the comment string. The <Set/Doc ID> must be either a query number, a query set name, or a document number enclosed in square brackets ([]). The comment string must follow the rules described in Section 2.7.1. Examples of COMMENT statements are:

```
COMMENT 3 "SOME COMMENT STRING"
```

```
CO [ 19] "VERY GOOD PAPER ON "FRABBLEGIBBETS""
```

The first example merely attaches the comment

SOME COMMENT STRING

to query set #3, while the second attaches the comment string

VERY GOOD PAPER ON "FRABBLEGIBBETS"

to document #19.

2.7.6 DELETE STATEMENT

The DELETE statement is used to delete query sets and/or comments from the user file area. Once a query set has been deleted it cannot be referred to in a MAKE statement or a From Clause, but it no longer takes up space in the users' file. Since each user is assigned only one cylinder of disk, it is important to remove unwanted query sets and comments when they are no longer needed.

The DELETE statement has three forms. The first looks like this:

DELETE <Set List>

"DELETE" is the keyword, and may be abbreviated "DEL". The <Set List> is a list of query set names and query set numbers of query sets that the user wishes to have deleted. This will remove both the query set and all associated comments. Examples are:

DELETE 3,7,JUNKSET

DEL BADSET

The second form is:

DELETE COMMENTS <Set/Doc List>

where "DELETE COMMENTS" is the keyword informing EUREKA to remove all user comments attached to the query sets and/or documents that make up the <Set/Doc List>. The keyword may be abbreviated "DEL COMMENTS". The query sets in the list may be referred to by either query number or by query set name, while the documents must be referred to by a list of document numbers separated by commas and enclosed in a single set of square brackets.

When this form of the DELETE statement is used only the comments attached to a query set or document are deleted, so the query sets may be referred to in later statements (but the comments are no longer available). Examples are:

DELETE COMMENTS 3,[15,23,5],GOODSET

DEL COMMENTS [18]

Note that users may delete comments from documents, but are not allowed to delete the actual documents.

The third form of the DELETE statement is:

DELETE MACRO <MACRO LIST>

As one would expect, this form is used for deleting macro definitions. "MACRO" is the keyword informing EUREKA that the following list is a list of macro definitions to be deleted

rather than a list of query sets to be deleted. Note that this command may be abbreviated to "DEL MACRO <Macro List>".

If the user wishes to delete all or most of his/her macros, query sets, and/or comments, he may specify that he wishes EUREKA to delete all sets/macros except the ones he wishes to save by putting the names and/or numbers of the queries/macros he wishes to have saved in the <Set List>/<Macro List> and precede the <Set List>/<Macro List> with the keywords "ALL EXCEPT".

Similarly, "DELETE ALL", "DELETE COMMENTS ALL", AND "DELETE MACRO ALL" delete all query sets, comments, and macros respectively, with no exceptions.

Examples of the use of "ALL" are:

DELETE ALL EXCEPT 5

which deletes all query sets and comments except query set #5.

DEL COMMENTS ALL

Which deletes all user assigned comments.

DELETE MACRO ALL EXCEPT BIGMAC

which deletes all macro definitions except the one named "BIGMAC".

2.7.7 LOGON STATEMENT

The LOGON statement is used to identify the user to the EUREKA system so that it can retrieve the user files and initialize a workspace for the user. The form of the LOGON statement is:

LOGON <User ID>

where "LOGON" is the keyword (it may not be abbreviated), <User ID> is the (up to) six letter identification code assigned to each user. If a person does not have a user ID, he may still use all facilities of the system except the storage of results between terminal sessions. If a user enters a query without first typing in a "LOGON" command, he is automatically logged in as a public user and allowed full access to the system. However, as soon as the public user logs off the system all of his query sets and comments are erased in order that the next public user may start with a clean slate. Examples are:

LOGON A34KR7

LOGON FRED

2.7.8 LOGOFF STATEMENT

The LOGOFF statement is used to log out from the system after a session. It tells EUREKA to save all the user files and free the user's workspace. Its form is:

LOGOFF

and there are no variations on its form.

2.7.9 PRINT STATEMENT

The PRINT statement has three uses. It may be used to print all or part of any document. It may also be used to print information about previous queries and their related query sets. Another use is to print macro definitions.

The form of the PRINT statement used for printing query set/statement information is:

PRINT <Set ID 1> TO <Set ID 2>

where <Set ID 1> and <Set ID 2> are either query numbers or query set names. This will cause the following information to be printed for each query set with a query number between that of <Set ID 1> and <Set ID 2>: query number, query set name (if present), query text, list of all documents making up this query set and their relative rank, and any comments associated with this query set. If "TO <Set ID 2>" is omitted, only the information for the set specified by <Set ID 1> is printed.

Examples are:

```
PRINT 3 TO LAST
```

```
PRINT JOE
```

and

```
PRINT 3
```

The form of the PRINT statement used for printing all or part of documents is:

```
PRINT <Context List> FROM <Set/Doc ID>
```

where <Context List> is the list of context items that the user wishes to have printed. Any list of context terms is valid here, as long as they are meaningful. If "PARAGRAPH" or "SENTENCE" is specified here, EUREKA looks at the query statement that generated the set list from which we wish to print information and then prints all paragraphs or sentences containing the search terms specified by the query statement. Therefore, it is not meaningful to command EUREKA to print a sentence from a set created by a MAKE statement, since there are no search terms in the MAKE statement to search for.

If no <Context List> is specified, the default value assumed is "DOCUMENT". <Set/Doc ID> may be either of the following:

- 1) Set name or query number;
- 2) List of document numbers separated by commas and enclosed

by square brackets ("[" and "]").

The command:

"PRINT <Context List> FROM <Set Name/Query #>"

will cause the portions of documents specified by the <Context List> of each document in the set list of that query to be printed.

The command:

"PRINT <Context List> FROM [Doc#1,Doc#2,...,Doc#N]"

causes the specified (by the <Context List>) portions of documents numbered "Doc#1", "Doc#2,...", etc. up to Doc#N to be printed.

All output from a PRINT statement is routed to the users terminal, unless he ends the print statement with "ON LP", in which case all output is routed to the line printer.

Examples are:

PRINT JOE

which prints all information (as described above) about the query and query set named "JOE".

PRINT JOE TO 14

which prints the query set information for every query set with a query number between 14 and that of set "JOE" (whether "JOE" has a lower or higher number than 14).

PRINT FROM JOE ON LP

which prints the entire document text of every document in query set "JOE" on the line printer. *** WARNING! BE CAREFUL WHEN USING THIS COMMAND, AS IT CAN EASILY GENERATE IMMENSE AMOUNTS OF OUTPUT ***.

PRINT TITLE FROM 3

which prints the title of every document in query set #3.

PRINT TITLE, AUTHOR FROM [3]

which prints the title and author of document number 3.

P FROM [3,43,22] ON LP

which directs EUREKA to print documents 3,43, and 22 on the line printer. Notice that "PRINT" may be abbreviated by "P".

P SEN FROM NEWSET

which prints every sentence in each document in the query set "NEWSET" that contains a term from the Term Expression from the FIND statement that generated "NEWSET".

PRINT COMMENTS FROM [12]

which prints all user-assigned comments attached to document #12.

P COMMENTS FROM 12

which prints all comments the user has attached to any document in query set #12.

The third use of the PRINT statement is printing macro definitions. The form used is:

```
PRINT MACRO <Macro Name>
```

"PRINT MACRO" is the keyword specifying that the following word is to be taken to be the name of a user defined macro definition and that this macro definition is to be retrieved from the user file and printed. If the word "All" is substituted for <Macro Name> then all macros and their definitions are listed. The output may be routed to the line printer by adding "ON LP" to the end of the command. Some examples are:

```
PRINT MACRO BUGS
```

which causes the definition of the macro "BUGS" (see Section 2.7.2) to be printed on the terminal.

```
P MACRO BIGMAC ON LP
```

which causes the macro named "BIGMAC" by the user to be printed on the line printer.

```
P MACRO ALL
```

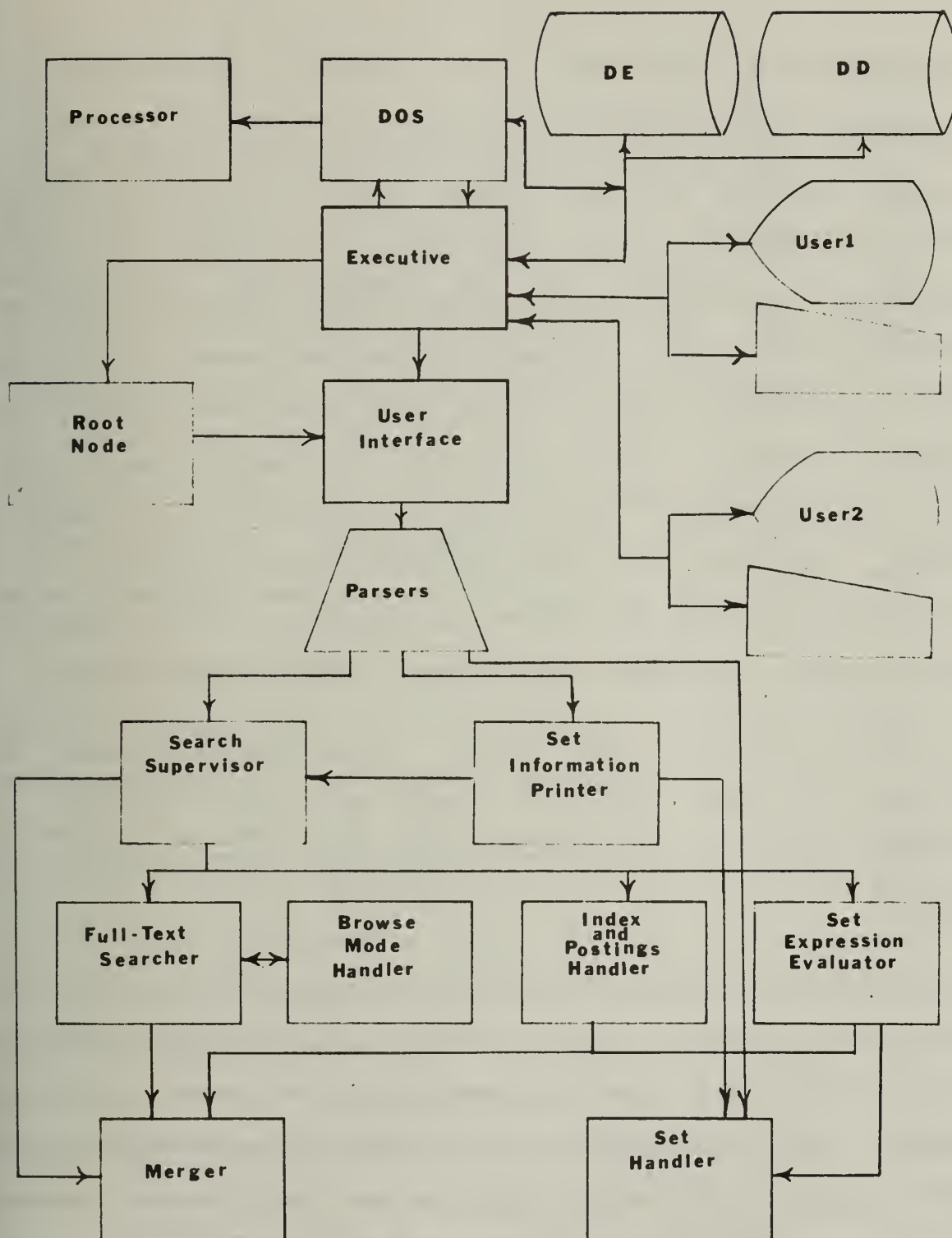
which prints out all macro names and the macro text associated with each macro name.

SYSTEM PROGRAMMERS GUIDE

3.1 AN OVERVIEW OF EUREKA

In order to obtain an overview of EUREKA before being faced with the gory details, let us examine a block diagram of the system structure (Fig. 3.1.1). This diagram shows the structure of EUREKA at a task level. The block labelled "Processor" is the actual PDP-11 hardware, which is allocated and controlled by "DOS", the DEC operating system, and by "EXECUTIVE", the EUREKA operating system. "DD" and "DE" are the two Diva 2314-style disk drives where both system and user files reside. "User1" and "User2" are the terminals and associated non-deterministic, non-rational physical cellular automata (1). Neither DOS nor EXECUTIVE shall be explained in detail here, as sufficient documentation exists elsewhere [1,2]. One point we should consider before proceeding, however, is the interrelationship of the two operating systems. DOS, the standard DEC operating system, is used primarily as a bootstrap and low-level software resource by EXECUTIVE, which actually provides almost all of the multi-user scheduling, allocation, and management facilities. All I/O requests, task startup and control, and memory management

(1) sometimes referred to as "humans".



SYSTEM FLOW
DIAGRAM
FIGURE 3.1.1

are done by traps to EXECUTIVE.

Each module is actually an invocation of a collection of one or more object modules that are logically grouped together and may be treated as a single unit by the executive for purposes of memory management and process control. Within tasks, JSR's and JMP's may be used to transfer control between modules, but between tasks, the EXECUTIVE trap \$PRFRM must be used in order to maintain EXECUTIVE's task control. Modules within a task share a common memory area, known as the "workspace", which must be allocated by the EXECUTIVE trap \$ALOC. This memory area (and any other memory allocated by a task) may be accessed by any routine in the task and by any routines in tasks initiated via \$PRFRM traps by the task, but by no others.

The Root Task or Root Node (ROOT) is an initialization and housekeeping task used to initialize user and system files, etc. It is the first module to be performed by EXECUTIVE upon starting up the system.

User Interface (USRNTF) is the window between the users and the internal EUREKA routines. It acts as a terminal handler and message router by accepting and formatting commands from the user, performing the Parser, and then displaying any error messages generated by lower-level routines and/or prompting the user for his next command. There is one invocation of User Interface in existence for each user

in the system at any given time. The Root Task starts up one invocation of the User Interface for each terminal attached to the system (this is determined by assembled-in constants in the code) at the time EUREKA is initialized. This invocation remains in existence for the duration of the execution (at least, until "SHUTDOWN" is typed in to shut EUREKA down). All EUREKA routines (except initialization routines) access user-dependent structures by way of pointers and tables passed to them by higher level routines. This allows EUREKA routines to be reentrant, simplifying greatly the process of adding more users to the system and minimizing memory usage since only one copy of the code need exist to serve all users.

The Parser (PARSER) decodes user commands by examining the command string typed in by the user. It creates either a Search Supervisor Table or a Set Handler Table that describes the services requested by the user and contains all the information needed by lower level routines to perform these services. The Parser then performs the correct action routine. If the command parsed requires either full-text searching or set list merging (FIND, MAKE, or PRINT <Context> FROM) Search Supervisor is performed upon completion of the parse. If the command parsed was PRINT <Set ID>, then the Set Information Printer (INFOPT) is performed. LOGON and LOGOFF are handled internally by the Parser. All other commands (CHANGE, DEFINE, COMMENT, and DELETE) cause the Set Handler to be performed.

Upon completion of the action routine, control is returned to the Parser, which immediately returns control to the User Interface.

The Search Supervisor (SRCHSP) is primarily a sequencing routine that controls the operation of the EUREKA routines used in a search. The Merger (MERGE), Index and Postings Handler (IPHNDL), Full Text Searcher (FTSRCH), and Set Handler (SETHLR) are all used by the Search Supervisor to complete a search.

The Merger (MERGE) is used to merge lists of documents together in order to construct lists of documents meeting the conditions of a Boolean function specified by the user in his/her search command. It is hoped that eventually the Merger will become a manager for a hardware merge unit now under construction.

The Index and Posting Handler (IPHNDL) is used to evaluate one term at a time. It is given one search term by the Search Supervisor and produces a list of documents in which this term appears. Note that in the case of a term containing several tokens, i.e. 'FULL TEXT', the Index and Postings Handler returns a list of documents containing both the words "FULL" and "TEXT" with no assurance that the string "FULL TEXT" actually appears. In this case, the Index and Postings Handler marks each document in the list by setting the full-text search bit in its descriptor (described in Sec. 3.8); The Full Text Searcher must be used to determine if the

words actually appear in the correct relationship within the documents listed by the Index and Postings Handler.

The Full Text Searcher (FTSRCH) performs the actual comparison of search strings from the user's command to the text of documents whenever a full-text search must be performed.

The Set Handler (SETHLR) performs all maintenance and accession of the user's personal files. All entries, deletions, and reads to/from this file must be done through this routine.

The Set Information Printer (INFOPT) is used to retrieve information on previous queries and/or macros typed in by the user. It uses the Set Handler to retrieve the desired information from the user's personal file, formats it, and then displays it on either the user's terminal or the line printer.

Now let us take a quick look at the information structures manipulated by EUREKA. There are effectively four types of information (excluding EXECUTIVE data) dealt with by EUREKA: the user's command string; the documents in the database and their associated accession mechanism; the user's Logon Block and personal file; and command tables passed from one task to another. The user's command string is entered by the user through the keyboard and is passed, along with a pointer to the user's Logon Block, to

the Command Parser. The Logon Block is effectively EUREKA's record of all system information specific to one user. The Command Parser then builds either a Search Supervisor Table or a Set Handler Table, depending on which action task is to be performed. If the command is MAKE, FIND, or PRINT FROM, then a Search Supervisor Table is constructed; otherwise a Set Handler Table is constructed. The table is then passed to the correct action routine (again, along with a pointer to the user's Logon Block). The action routines use the tables passed to them to determine what actions are to be performed (i.e. read a set list, search on a list of terms, etc.) and the Logon Block to find the correct user file to use and other such user-specific data. The document file and its associated accession mechanism (including the index, postings, and hash files) is used to perform the actual searches and to display text on the user's terminal. The user's personal file is used to store the record of his/her past searches, along with any macro definitions or comments attached to sets or documents by the user.

All the above tasks and information structures will be described in greater detail in the following sections.

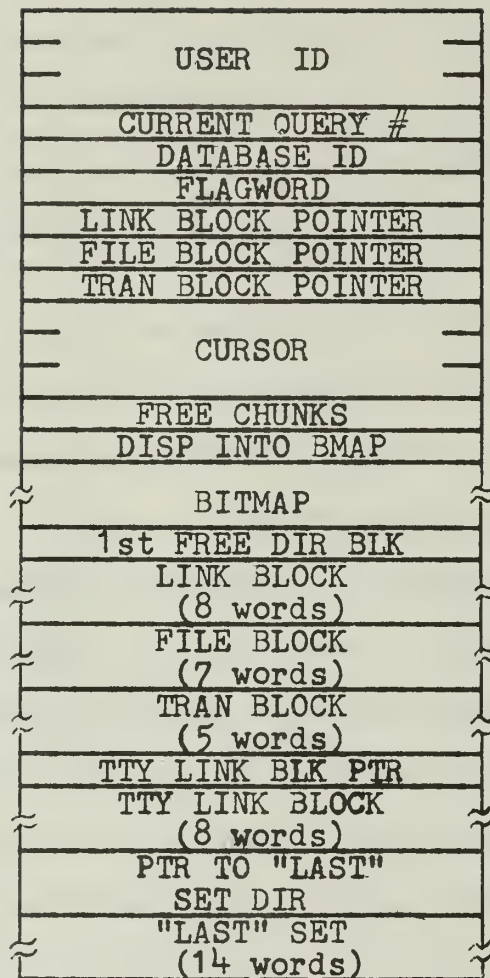
3.2 ROOT TASK

The first module we shall consider is the Root Task or Root Node (ROOT). This is a relatively uncomplicated module that essentially gets things started for the rest of the system and then closes up shop when the system is shut down.

The Root Task starts the system up by:

- 1) Calling (via a JSR) subroutine IRINIT, which opens all files except the user files, .INITs the terminals, and sets up some non-relocatable scratch spaces;
- 2) Performing n copies of the User Interface, where n is an assembled-in constant with global name LGNUM, passing each the address of a different Logon Block. The layout of the Logon Block is shown in Fig. 3.2.1;
- 3) Doing a TRAP 5 to wait until all n copies of the User Interface have executed \$RETN traps; i.e. SHUTDOWN has been typed in at all terminals).

Once all copies of the User Interface have died, the Root Task closes all relevant files and then executes a \$RETN trap to return control to the EXECUTIVE, thus shutting down the system.



USER LOGON BLOCK

Fig. 3.2.1

3.3 USER INTERFACE

The User Interface (USRNTF) is the user's window into the system. Each terminal has one invocation (task) of the User Interface associated with it (via the Logon Block passed to the task at the time it was initiated via a \$PRFM trap by the Root Task). This User Interface task handles query prompting, formatting, Parser invocation, error message display, and statistics recording for the user at its associated terminal and does not die until "SHUTDOWN" is typed in at the terminal.

The first action performed by the User Interface is the setting up of a buffer/statistics area. One component of this buffer is the text buffer that is passed to the Command Parser (and hence the rest of the system). Another large section of the buffer is the statistics area in which all timing and frequency statistics are recorded. Double buffering is used for the statistics area in order to decrease the number of I/O requests made during operation of the system. Once the buffers have been initialized, the User Interface enters a loop in which it:

- 1) Clears and resets the least recently used statistics block;
- 2) Re-initializes the byte count in the terminal I/O buffer header;
- 3) Reads the next query typed in by the user into the text buffer;
- 4) Checks to see if the query was too long;

- 5) Checks for continuation to another line, loops back to (2) if so;
- 6) Checks for "SHUTDOWN" having been typed in, goes to shutdown routine if so;
- 7) Performs the Parser, passing it pointers to the Logon Block and the query text string;
- 8) Displays error message (if any), or "COMMAND COMPLETE" message if no error has occurred when control is returned from the Command Parser;
- 9) Records statistics into buffer and writes block containing both buffers if both buffers are full;
- 10) Loops back to (2).

The shutdown routine writes the user statistics block out to disk (one buffer may be empty, depending on whether the user typed in an even or odd number of queries), and then executes a \$RETN trap, returning control to the Root Task.

3.4 COMMAND PARSERS

The Command Parser module consists of a collection of different routines that each parse one EUREKA command, plus several subroutines used to perform common functions. Access to the Command Parser module is through routine FIND, which parses the FIND, DEFINE, LOGON, and LOGOFF statements. FIND initially allocates

workspace for the entire Command Parser module (placing the address in R5), fills in the address of the beginning and end of the query text in the workspace, and then does a JMP to the correct Command Parser routine based on the first two or three letters of the command.

The individual Command Parser routines shall not be described here since they are very straightforward linear-scan, fill-in-the-table routines.

Once the Command Parser routine has constructed a command table of the appropriate type it then initiates via a \$PRFM trap either the Set Handler, Search Supervisor, or Set Information Printer, depending on instruction type.

As soon as the action routines execute \$RETN traps, returning control to the Command Parser, the Command Parser executes a \$RETN trap, returning control to the User Interface in order to begin the cycle again.

3.5 SET HANDLER

The next task we shall consider is the Set Handler (SETHLR), which maintains the users' personal files. All changes to the user file or retrievals therefrom must be made by this task in order to

maintain the integrity of the user file structure.

3.5.1 USER FILE STRUCTURE

Each user in the EUREKA system is assigned a personal disk file in which all user-specific records are stored. Since user information is dynamic and of varying lengths and types, we must have an access/storage system that can cope efficiently with rapidly changing, non-homogeneous data. This system should also seek to minimize the number of disk accesses required by common functions, as they are currently one of the more troublesome bottlenecks in EUREKA.

The data structure chosen for this task is shown in Fig. 3.5.1.1. It exists in the medium of a disk file consisting of one cylinder of disk. This gives us 240 contiguous blocks of 256 16-bit words. Since the blocks are allocated in one cylinder they may all be accessed without moving the heads of the disk unit, thus avoiding some seek time on sequential reads. The file is accessed in relative (.BLOCK) mode, giving us a block address space of 0-239 and a byte address space of 0-511. User information is stored in blocks 7-239, with blocks 0-6 being used as directory space. The storage blocks (7-239) are divided (for allocation purposes) into chunks of 64 bytes (8 per block). File space is allocated in chunks starting

USER FILE STRUCTURE

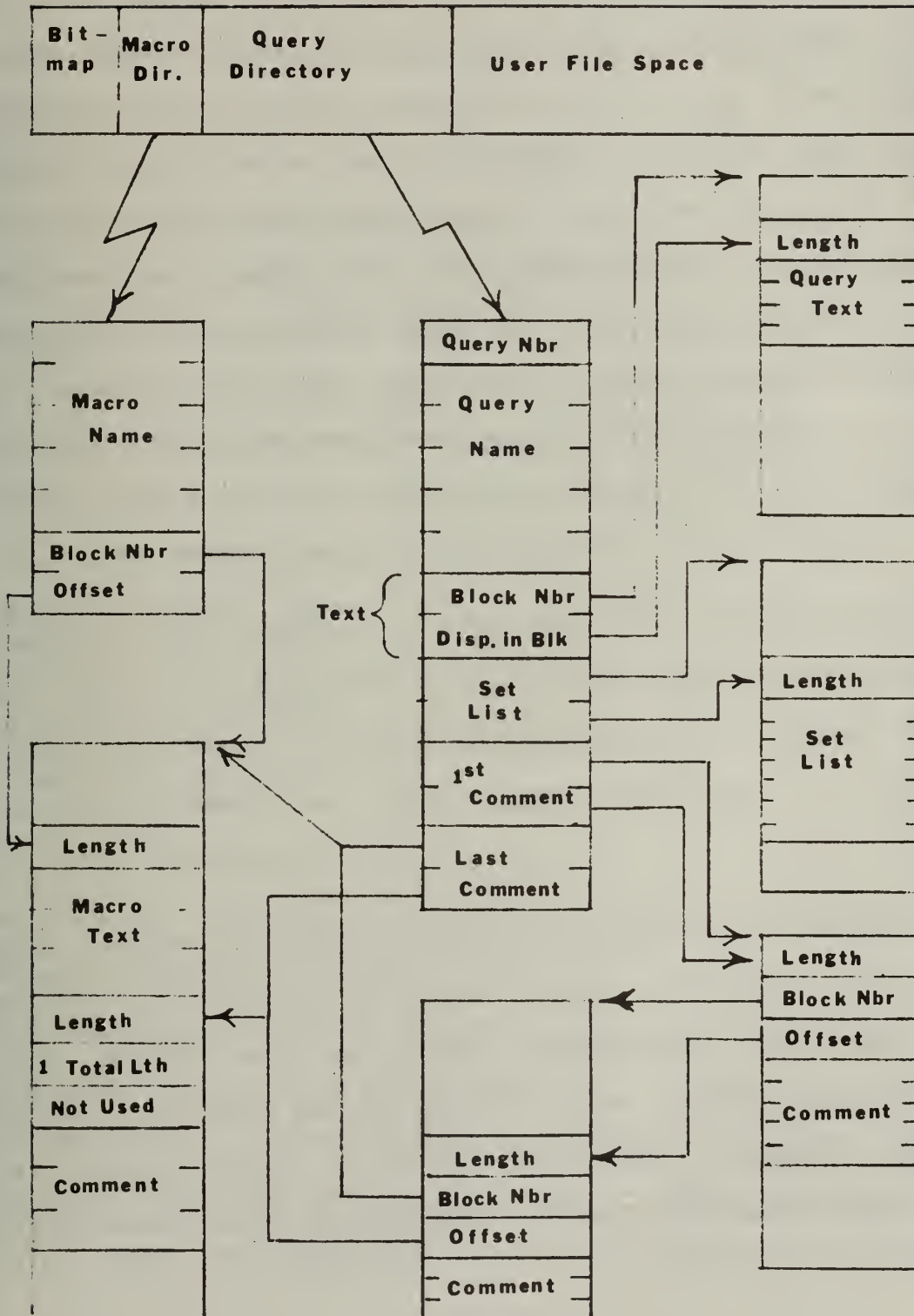


Fig. 3.5.1.1

at the last block (239) of the file and grows toward the front of the file. Chunks within each block are allocated from byte 448 proceeding back to byte 0. Since Set Handler routines requesting disk space from the Bitmap Handler are only given a starting address which is actually the lowest byte of the lowest block number of the contiguous space allocated them, only the Bitmap Handler (ALOC) need be concerned with this allocation pattern. The Bitmap Handler keeps track of which chunks of disk are in use by recording their status in a bitmap which occupies the last 240 bytes of the user's Logon Block when the user is logged in or the first 240 bytes (0-239) of block 0 of the users disk file when logged out.

In this bitmap, a bit value of "0" implies the corresponding chunk is in use, while a bit value of "1" implies that the chunk is free. The mapping scheme that allows us to associate bits with chunks will be discussed along with the Bitmap Handler in Sec. 3.5.8.

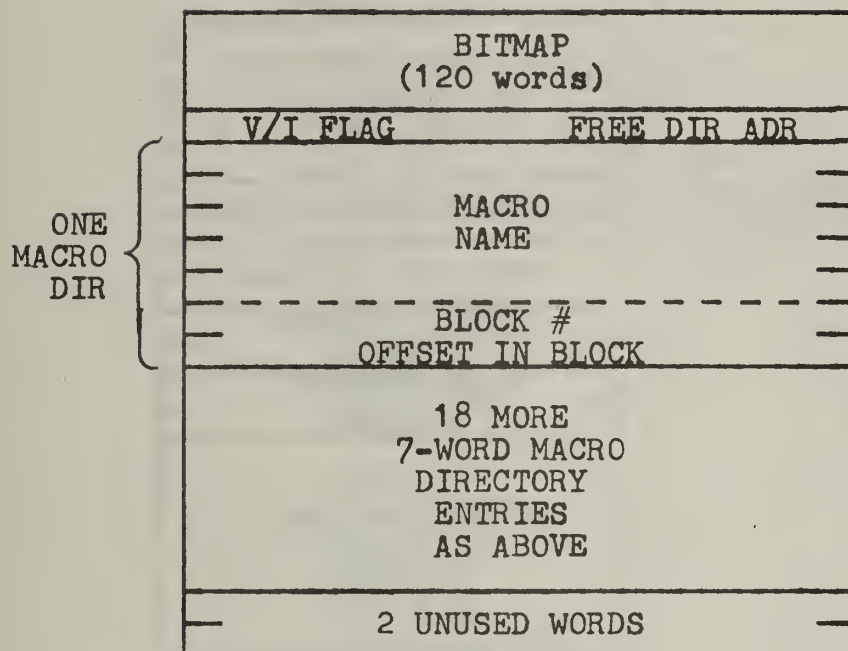
The rest of the first block (block 0) of the user's file is occupied by a one-byte (byte 240) free directory block number, a one-byte (byte 241) valid/invalid flag for the bitmap, and the user's macro directory in words 121-253 (bytes 242-507). See Figure 3.5.1.2 for details. The free directory block number is the relative block number of the first directory block (block 1-6) that

has an unused entry in it. The valid/invalid flag is used to prevent users from accidentally wiping out information in their files by logging back on after a system crash or other calamity occurring while they were logged on has destroyed the copy of their bitmap in their Logon Block before it could be rewritten to disk. Whenever a user logs on, the bitmap is read in from their disk file and stored in their Logon Block and the valid/invalid flag is set to "-1" as a flag that the bitmap is no longer current. When the user logs back out, the updated bitmap is transferred from the user's Logon Block back to the first 240 bytes of block 0 of the user's file and the valid/invalid flag is set to zero to indicate that the bitmap is current again. Should the system crash while the user is logged on, the bitmap must be rebuilt by the off-line routine BITFIX which builds a new bitmap for the user based on the current contents of the user's file.

Words 121 through 253 of the first block are taken up by the user's macro directory. The macro directory consists of 19 7-word blocks structured as in Fig. 3.5.1.2. A 5-word (10 character) macro name is followed by the starting address (block number and offset within block) of the macro text. At this starting address will be found one word containing the length of the macro text in characters, followed by the macro text. This directory (and the entire file) is maintained by the Set Handler routine ALOCD, the

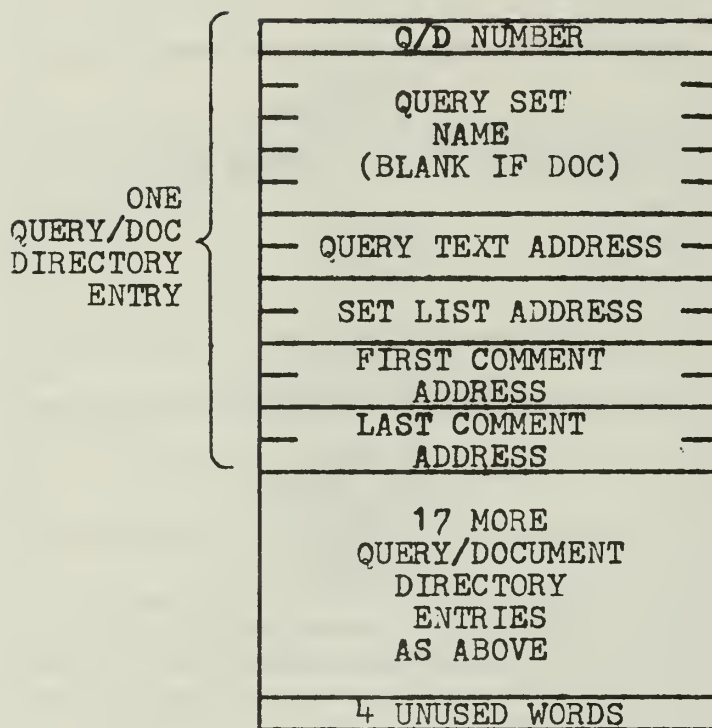
Bitmap Handler, which uses the first word of each macro directory as a flag for allocation of directory slots. If the first word of a directory slot contains zero (in binary, not the character), that directory slot is free; any other value shows that the directory slot is in use as a pointer to some macro text.

The next six file blocks (1-6) are occupied by the query set directory, as shown in Fig. 3.5.1.3. This directory is structured much like the macro directory, the main difference being more fields within each directory slot. Each query set directory slot can be seen to be a 14-word long block containing a query set number, query set name, and pointers to the starting addresses of all of the pertinent information on disk that makes up the query set. Both the query text and query set list are stored in the same manner as the macro text (length word followed by information). The comments are slightly more complex, as they are stored as a one-way linked list so that comments may be added to existing queries or documents. Refer to Fig. 3.5.1.4 for details of the comment chain structure. For comments, the field labeled "1ST COMM PTR" points to the first comment in the chain. Each comment consists of a length word followed by a two-word link field (block number and displacement) that points to the next comment in the chain. The comment text follows immediately after the link. The last comment in the chain is pointed at by the field of the directory entry labeled



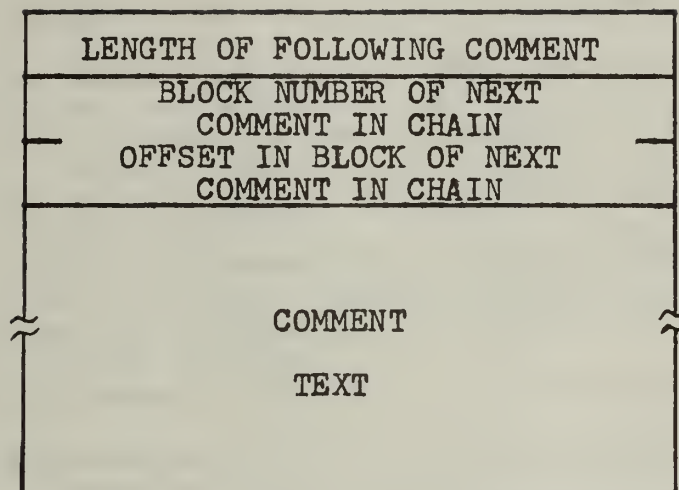
BLOCK 0 DETAIL

Fig. 3.5.1.2



BLOCK 1-6 DETAIL

Fig. 3.5.1.3



COMMENT DETAIL

Fig. 3.5.1.4

"LAST COMM PTR" and is also flagged by having the high-order bit in the first word of the link field set to 1. Also stored in this word is the total length of all comments (retrieved by setting bit 15 to 0). The tail pointer in the directory entry is used to speed up the attaching of comments by allowing the File Writer routine to avoid running down the chain each time a new comment is to be added. The bit flag is used to signal the end of the list to routines that are running down the list, such as the File Reader or the Delete Routine. The total length of all comments field is used to put an upper bound on the length of the buffer size needed to read in user comments. If a query has no comments attached to it, both of its comment pointers are set to "-1" to flag their non-existence. Document directories occupy the same slots as query slots in order to avoid having three kinds of directories. A document directory entry is distinguished from a query directory by having bit 15 of the query/document number field set to 1. Again, in order to retrieve the document number one must clear this bit. The only fields in a document directory entry that are meaningful are the query/document number as just discussed, and the comment pointers. The comment chain attached to a document is stored in exactly the same form as one attached to a query set. As in the case of the macro directory the first word of the query/document directory slot is used as a flag word. If this word contains a negative value,

then it is a document directory; if it contains a positive number, it is a query directory; if it contains zero, it is an unused directory slot.

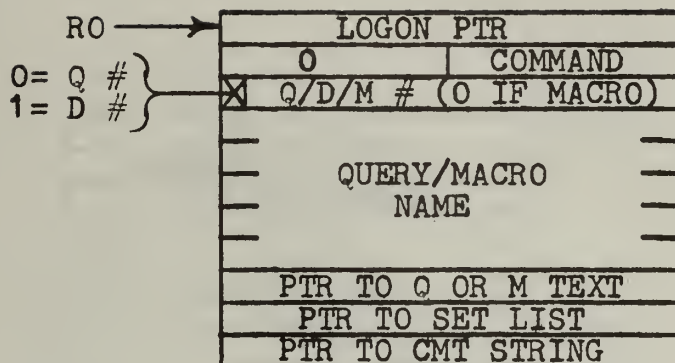
3.5.2 SET HANDLER TABLE

The Set Handler may be invoked by the Parser when processing CHANGE,DELETE, or COMMENT statements; the Search Supervisor when processing MAKE or FIND statements; and the Information Printer when processing PRINT commands. These tasks use the EXECUTIVE trap \$PRFRM to start up the Set Handler.

These routines communicate with the Set Handler via a table known as the Set Handler Table. The address of the Set Handler Table must be placed in register R0 by the performing task. There are effectively three different kinds of Set Handler Tables, as shown in Figs. 3.5.2.1, 3.5.2.2, and 3.5.2.3. The table shown in Fig. 3.5.2.1 is used for all read and write operations (op codes 0-6,12,13,17). The contents of the read/write form of the Set Handler Table are as follows:

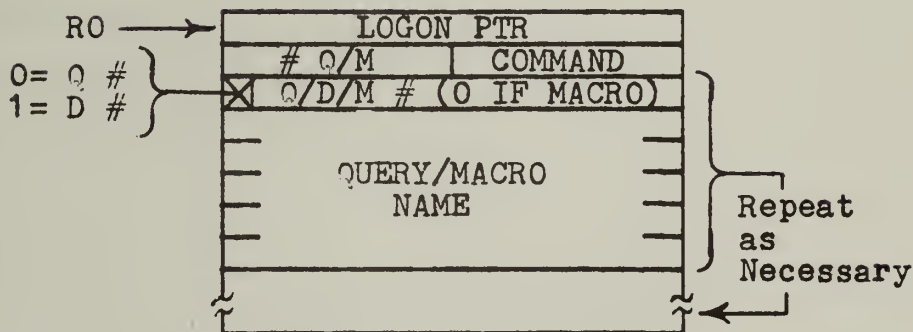
<u>OFFSET</u>	<u>CONTENTS</u>
0-1.....	address of Logon Block
2.....	command (one byte long only)
3.....	not used
4-5.....	query/document # - high-order bit is set to 0 if query #, 1 if document. this word is set to 0 for a macro read or write.
6-15.....	query/macro name
16-17.....	address of query/macro text
18-19.....	address of set list
20-21.....	address of comments

A list of Set Handler commands is given in Table 3.5.2.4. Note that not all fields of this table will be filled in on every invocation of the Set Handler. Read and write commands need fill in only those addresses pertaining to the items to be read/written (there is one exception, however; see the section describing FILRDR). For instance, if a command of "2" (read query text and set list) is used, then only the addresses of buffers in which to store the query text and the set list need be allocated. Similarly, if a command of "6" (write comments) is used, then only the address of the buffer containing the comments to be attached to the specified set need be filled in. Also, it is not necessary to fill in both the set name and set number on a read, either one being sufficient. The address of the Logon Block, a command, and either a set name or set number must always be present, however. In order to avoid accidental matches to incorrect sets in the directory search, care should be taken to clear unused set name/number fields when only one of the two is being used. When not being used, the query/macro name should



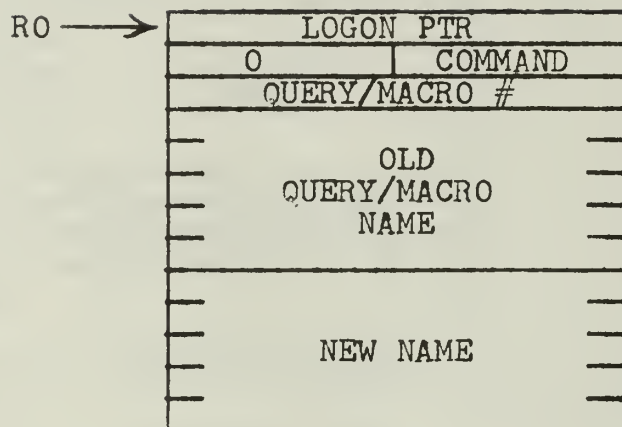
READ/WRITE FORMAT
(cmds 0-6, 14, 15)

Fig. 3.5.2.1



DELETE FORMAT
(cmds 7-12, 16, 17)

Fig. 3.5.2.2



RENAME FORMAT
(cmds 13, 20)

Fig. 3.5.2.3

<u>ACTION TO BE PERFORMED</u>	<u>COMMAND</u>
READ QUERY TEXT.....	0
READ SET LIST.....	1
READ QUERY TEXT AND SET LIST.....	2
READ COMMENTS.....	3
WRITE QUERY TEXT AND SET LIST.....	4
WRITE QUERY TEXT, SET LIST, AND COMMENTS.....	5
WRITE COMMENTS.....	6
DELETE FOLLOWING QUERY TEXT, SET LIST, AND COMMENTS.....	7
DELETE ALL QUERY TEXTS, SET LISTS, AND COMMENTS EXCEPT FOLLOWING.....	8
DELETE FOLLOWING COMMENTS.....	9
DELETE ALL COMMENTS EXCEPT FOLLOWING.....	10
RENAME QUERY SET.....	11
READ MACRO TEXT.....	12
WRITE MACRO TEXT.....	13
DELETE FOLLOWING MACROS.....	14
DELETE ALL MACROS EXCEPT FOLLOWING.....	15
RENAME MACRO.....	16
READ LIST OF ALL COMMENTED DOCUMENTS.....	17
NOT USED.....	18
READ LIST OF MACRO IDENTIFIERS.....	19
READ QUERY NAME/NUMBER ONLY.....	20

Set Handler Command Codes

Figure 3.5.2.4

be set to blanks, and the query/document number should be set to zero.

The next type of Set Handler Table we shall consider is the delete form of the Set Handler Table (Fig. 3.5.2.2), used only when deleting query sets, comments, or macro definitions (command codes 7-10,14,15). Its contents are the same as the read/write form except for several minor variations: Byte three contains the number of items to be deleted, unless a code meaning "delete all except" (codes 8,10,15) is used, in which case the third byte contains the number of set/macro identifiers that are to be saved. Note that in this case a zero in byte three implies "delete all". The only other difference from the read/write form is the absence of buffer addresses. These are replaced by a series of six-word long blocks containing a query number and/or name for each query/macro to be saved or deleted, depending on the command used. On a "delete" command (codes 7,9,14) the identifiers of the sets/comments/macros to be deleted are listed, while in the case of a "delete all except" (codes 8,10,17) the identifiers of the items to be saved are listed.

The last Set Handler Table to consider is the rename format (Fig. 3.5.2.3). This table is used only for rename commands (codes 11 and 16) and has a very simple layout. The first three words of the table are identical with the read/write form (except for command

code, of course). These three words are then followed by the old query/macro name (5 words) and the new name to be assigned to the query/ macro (5 words).

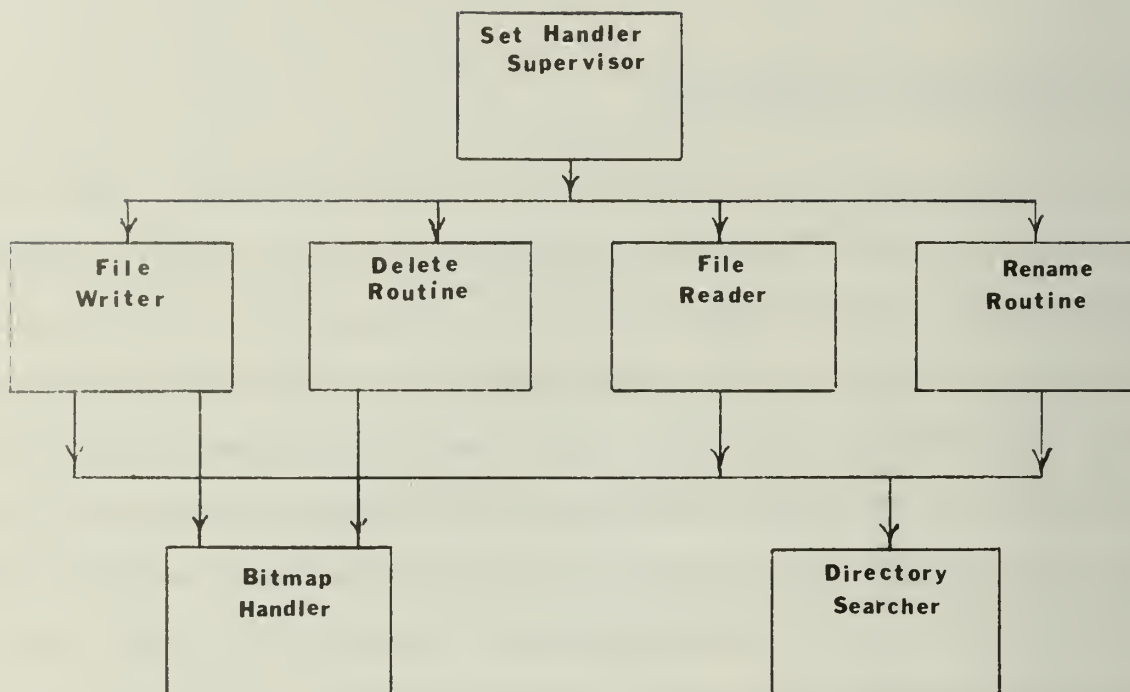
3.5.3 SET HANDLER SUPERVISOR

Now that we have an understanding of the input to this module, let us consider the routines contained in the Set Handler module and delineate their individual functions. Figure 3.5.3.1 shows us the various routines in the Set Handler and their interconnections. Notice that the only entry path into the Set Handler is through the Set Handler Supervisor (SETHLR). This routine of the Set Handler module acts as a startup routine for all the action routines within the Set Handler. Its functions are:

- 1) Allocation of workspace for all routines;
- 2) Translation of all requests for action on the "last" set into a specific set name/number;
- 3) Determination of which action routine to call (via a "JMP" command).

The sequence of operations performed by the SETHLR routine is as follows:

- 1) Check for valid command code;
- 2) Check to see if this is a reference to the "last" set (signalled by "LAST" in the query name). If "LAST" occurs in any set name



**SET HANDLER
SYSTEM FLOW
DIAGRAM**

FIGURE 3.5.3.1

- slot, then move in the name/number of the "last" set, if it exists;
- 3) Allocate workspace, put address in R5;
 - 4) Initialize the .TRAN Block in the LOGON Block (using the last 320 (500 octal) bytes of the workspace, starting at byte 72 (120 octal) as the I/O buffer);
 - 5) Do a JMP to the correct action routine.

No file accesses are made from this routine, the "last" set information being obtained from the LOGON Block. The only data structures modified by this routine are the LOGON Block (the .TRAN Block is set up and the "last" set name is changed on rename of "last" set) and the Set Handler Table (the "last" set identifier is moved in if the "last" set is referenced). For further information, refer to the program listing.

3.5.4 FILE WRITER

Now let us consider the File Writer (FILWRT). This routine accepts a Set Handler Table containing a "write" command and pointer to information to be written to disk in the user's personal file. This routine, the delete routine, and the logon/logoff routines are the only ones allowed to alter the user's file.

When entered (via a "JMP" from routine SETHLR), FILWRT expects to find the address of a Set Handler Table in register R0 and the address of its workspace in R5. This Set Handler Table should contain a command code of 4-6 or 13. A set identification, macro name, or document number must also be present, along with a pointer to the user's Logon Block. Last, but not least, there must be some pointers to the buffer(s) containing the information to be written to disk. If a query set is being written out to disk, then either a command of 4 or 5 will be used, depending on whether the user has attached comments to the set or not. The pointer to the query text and to the set list must be present in either case. The pointer to the query text is the address of a buffer containing:

- 1) A pointer to the Logon Block;
- 2) A pointer to the carriage return - line feed ending the text string;
- 3) The actual text of the query.

This strange format is due to the current User Interface/Parser communication protocol. The File Writer computes the length of the query text (without the carriage return - line feed) from the starting address of the text and the address of the CR-LF. When written to disk, the Logon Block pointer and CR-LF pointer are replaced by a word containing the length of the query text in characters (note that this is identical with the number of bytes of

storage). Similarly, the set list pointer points to a buffer containing:

- 1) The number of documents in the set (1 word);
- 2) Two words per document, the document number and its relative rank in the set list.

The number of documents is transformed into a length in bytes (i.e. multiplied by 4) before being written to disk in order to simplify the read mechanism. If comments are present, the comment pointer is the address of a buffer containing:

- 1) Two blank words for use by the File Writer as link words;
- 2) The length of the comments in bytes;
- 3) The comment string.

The output format of the comment string will be discussed along with the comment string writing mechanism.

When writing out a new query set, the File Writer first calls routine ALOCD, the disk file space allocator, to obtain a directory entry slot to use for the new query set. As soon as this request is granted, the query set identification number and name are moved into the area in the workspace reserved for building the new query directory entry. Next the query text length is computed and stored at the head of the text in the output buffer. Disk space for the text is then requested by another JSR to ALOCD. When the disk address is returned by ALOCD it is entered in the workspace copy of

the directory entry record and is passed (along with appropriate pointers to the text buffer, Logon Block, etc.) to the routine WRTDSK, a subroutine that formats the information into disk block size and writes it to disk.

The query set list and comments are then handled in a similar fashion. Upon completion of their transfer to disk the File Writer must then write the directory entry to disk and update the "last" set information in the Logon Block (since the new query set is by definition the new "last" set). When these duties have been completed, the File Writer performs the EXECUTIVE trap \$RETN, effecting a return of control from the Set Handler to the performing task.

Macro text writes (code 13) and comment only writes (code 6) are handled somewhat differently. A macro text write resembles a query set write. However, since neither a set list nor comments will be present the macro directory entry is shorter than the regular query set directory entry, leading to many directory size kludges in the File Writer (See Figs. 3.5.1.1 and 3.5.1.2 for a diagram of the directory layouts). Another important difference is the format of the text buffer passed to the File Writer. Unlike the query text, the macro text is passed in a buffer containing the character count of the text followed by the text itself. Since this

more closely resembles the format of the document list (at least after conversion of the document count into a byte count), the macro text is put in the correct format and written to disk by the same section of the File Writer that handles the set list.

Comment writes (code 6) are the most complex operations performed by the File Writer. Three distinct cases must be considered:

- 1) Adding comments to a query set that currently has no comments;
- 2) Adding comments to a document that currently has no comments;
- 3) Adding comments to a query set or document that has previously had comments attached to it.

We discover into which category a given request falls by examining the query/document number and the directory entry for the query set (or existence/non-existence of a directory entry in the case of documents). The subroutine DIRSRH is used to search the directory for this information. If the entity to which the comments are to be attached is a document and no comments have been previously attached then no directory entry will exist for this document. If the entity is a query set with no currently attached comments, a directory must exist, but the pointer to the comment string will be set to -1. For either a document or a query set with existing comments, the directory entry for the document or query set will contain pointers (disk addresses) to the head and tail of the comment chain for that

entity (see Fig. 3.5.1.4 for a diagram of the structure of the comment list). In light of this information, we can see how the File Writer must proceed. First, the directory searcher (DIRSRH) is called to find out if a directory entry exists for this entity and to retrieve a copy if it does (non-existence is flagged by the Directory Searcher by moving -1 to the first word of the buffer in which it has been requested to place the directory copy).

If the entity is a document and no directory entry exists, the File Writer performs the following actions:

- 1) Requests a directory slot and disk space for the comments from ALOCD;
- 2) Fills out the directory entry with both the head and tail pointers pointing to the newly allocated disk space in which the comments are to be written;
- 3) Moves "-1" to the first word of the output buffer for the comments (again, see Fig. 3.5.1.4 for the comment chain layout) to flag the non-existence of further links in the chain;
- 4) Moves the length of the comment string to the second and third words (total comments length and local string length, respectively);
- 5) And then writes out the directory entry and comment string.

Attaching comments to a query set that currently contains no comments is done in a similar fashion, altered only by the pre-existence of a directory entry that must be altered rather than

allocating a new directory entry slot. In the case of adding a new comment to a query set or document that already has one or more comments attached to it, the File Writer must:

- 1) Read the comment pointed to by the tail pointer of the directory entry;
- 2) Move the word containing the total length of the existing comments (second word of old last comment) to the second word of the new tail comment and add in the length of the new comments;
- 3) Move the disk address of the new comments to the link field of the old tail comment and rewrite the old tail comment (or the first block thereof if it extends across a block boundary);
- 4) Update the tail pointer in the directory record to point to the new comment;
- 5) Write out the new comment record;
- 6) Rewrite the updated directory record;
- 7) Check to see if the "last" set has been modified, and update the "last" set information in the Logon Block if so.

3.5.5 FILE READER

Now that we have seen the mechanism used for writing data into the users' files, let us consider the mechanism used for retrieving said data. This routine of the Set Handler is called the File

Reader (FILRDR). It is entered by a JMP command from the SETHLR routine whenever SETHLR detects a read op code (0-3,11-14) in the Set Handler Table. For each of these op codes, the Type I Set Handler Table is used. The only other routine referenced by the File Reader is the Directory Searcher (DIRSRH).

The first operation the File Reader performs (after a minute amount of housekeeping) is to determine what is requested of it. If a list of macros or a list of all comments is requested, then special sections of the File Reader are JMP'ed to (MACALL and COMALL, respectively). If a macro text or any type of set information is requested, then FILRDR must obtain the directory entry for the data to be read. This is done by one of two methods; if the data requested is anything but information from the "last" set, then the Directory Searcher (DIRSRH) is called via a JSR command, or if the information desired is from the "last" set (signalled by a set name of "LAST" in the Set Handler Table), then the section of the File Reader following label LASTRT is used to obtain the information directly from the user's Logon Block, thus avoiding one or more disk reads. In either case, the address of the disk block containing the disk directory for the set/macro to be read is returned, along with the directory entry itself (in working storage), to the File Reader. If the read requested was from the "last" set, but no "last" set exists, then a "-1" is put in the

first word of the Set List Buffer and a \$RETN trap is executed in order to allow a "read last set" to be done on the first query of a session without causing undue problems. Note that this forces us to fill in the set list pointer of all Set Handler Tables referencing the "last" set and check for this error condition in each routine doing a read from the user's file, even if we are not requesting a set list read. If a read is requested from any other non-existent set than the "last" set, an \$ERROR trap return is done. Once we have the directory entry for the selected set or macro, then we may begin to read in the requested information. The query text, set list, comments, and macro definition reads are all done by essentially the same section of code (starting at label READIT) with parameters for the read loop set to point to the correct buffers, etc. by a compare-branch tree preceeding the loop for each iteration of the loop. The code starting with label DIDIT is a trailer section that follows a performance of the read loop and determines whether another loop through the read loop is needed. This occurs whenever both the query text and set list must be read for a set or when comments are being read and the one just read has a pointer to another comment chained to it. Once all the information has been read in, the File Reader exits via a \$RETN trap at label DONE.

The two special case sections of the File Reader, MACALL and COMALL, are straightforward linear search strategies that merely obtain the desired lists of macro/document ID's. Macros are listed in the buffer pointed to by the query text pointer with one macro name every five words and the number of macro names stored in the query number field of the Set Handler Table.

Commented document lists are returned in the set list buffer, with the first two words of the buffer being identical counts of the number of document numbers following. After the two count words come the document numbers themselves. These are in the form of two word entries, the first word being the document number and the second zero in order to simulate a normal set list.

This concludes our discussion of the File Reader and allows us to proceed to the next major routine in the Set Handler Module, the Delete Routine (DELRTN).

3.5.6 DELETE ROUTINE

The Delete Routine (DELETR) handles requests to delete query sets, comments, or user-defined macros from a user's file. It does this by finding the directory entry for the information to be deleted, zeroing the first word of the directory entry if the request is to delete a query set or a macro or by moving "-1" to the

comment pointers if comments are to be deleted. After this has been performed the Delete Routine calls the Bitmap Handler (ALOCB) which frees disk space involved.

In finding the directory entry for a query/macro to be deleted, the Delete Routine reads in the the user's directory (one block at a time) and scans each block linearly. This approach is used rather than calling the Directory Searcher to locate the directory in order to minimize the number of disk accesses when the user is deleting a large number of queries at once. Each query/macro directory entry scanned is compared to the list of queries/macros attached to the Set Handler Table passed to the delete routine by the Delete Statement Parser. If the query/macro identifier matches one of the identifiers in the list of queries/macros attached to the table and a "delete" command code (7 for query set, 9 for comments, 14 for macro) has been entered in the command table; or if no match has been found in the identifier list for a directory entry and a "delete all except" command code (8 for query sets, 10 for comments, 15 for macros) has been entered in the command table, then the directory address is passed to the portion of the Delete Routine which causes storage deallocation and directory deletion; otherwise, the directory entry is left unaltered and the search continues. It should be noted that "delete all" for query sets, macros, or comments is denoted by a "delete all except" command with

a null list of identifiers.

Disk storage deallocation for query set lists, query texts, and macro texts is done by a loop that reads in the information to be deleted in order to get the length of the disk space to be freed and then calls the Bitmap Handler (ALOCB) which marks the space free in the user's bitmap. Comments are handled in a similar fashion, but by a recursive routine (CHASER) which runs down the links of the comment chain marking the disk space occupied by each comment free in the bitmap.

After the disk space has been marked free in the bitmap the comment pointers in the directory are set to -1 if comments only have been deleted or the first word of the directory entry is set to zero if an entire query set/macro definition has been deleted.

After all items in the query/macro identifier list have been deleted or the end of the directory has been reached, control is returned to the Delete Statement Parser (and hence to the User Interface) via a \$RETN trap.

3.5.7 RENAME ROUTINE

The Rename Routine (RENAME) of the Set Handler module is an extremely simple routine. Its purpose is to attach a (new) mnemonic name to a query set or to a user defined macro.

The Rename Routine accepts as input in register R0 the address of a Type III Set Handler Table (see Section 3.5.2). This table contains the current identifier (name or number) of a query set or user macro and the new name to be attached to the set/macro. The Rename Routine uses the Directory Searcher (DIRSRH) to retrieve the address of the directory entry for the set/macro to be renamed. The Rename Routine then reads in the disk block containing the directory entry, replaces the name field of the directory entry with the new name from the Set Handler Table, rewrites the block containing the directory entry, and exits via a \$RETN trap.

3.5.8 BITMAP HANDLER

The Bitmap Handler (AL OCD) is used to allocate/deallocate disk storage and directories for the Set Handler. This routine is performed as a subroutine (via a JSR) by the File Writer and the Delete Routine and receives all of its parameters through the Set Handler Workspace, whose base address is located in register R5 throughout the Set Handler. The workspace parameters used by this

routine are the Logon Block pointer (LOGPTR), amount requested (NBRREQ), block number (RELBLK), and offset in block (BLKDSP). The exact layout of the Set Handler Workspace is stored as a template macro in the file SYSMAC.SML.

The NBRREQ field of the workspace is used to hold the number of bytes of storage to be allocated/deallocated or a directory allocate/deallocate flag when either query or macro directories must be manipulated. The Logon Block pointer contains the address of the user's Logon Block, needed by this routine for reading directories from the user's file when doing directory allocates/deallocates. The block number and offset in block fields are used for passing disk addresses of the beginning address of disk space allocated or to be deallocated by the Bitmap Handler.

There are essentially five cases of Bitmap Handler action requests to consider:

- 1) Allocating user file space;
- 2) Deallocating user file space;
- 3) Allocating macro directories;
- 4) Allocating query set directories;
- 5) Deallocating query set/macro directories.

Deallocation of directories, whether they are macro or query set directories, is very straightforward. The Bitmap Handler is passed

the disk address of the offending directory entry in the disk address fields of the workspace and needs merely read in the required block, move zero to the first word of the entry, rewrite it, and return. Allocation of directories is only slightly more complex. When requesting allocation of a directory, the calling routine (FILWRT) places a flag describing the type of directory desired in the NBRREQ field of the workspace (these flags, MACMSK and DIRMSK, are contained in the file SYSMAC.SML). The two short routines MACALC and DIRALC are used to allocate macro and query set directories, respectively. Both routines work by searching linearly through the proper directory until they find a directory slot with a first word of zero or the end of the directory. If a free slot is found its block number and offset within the block is placed in the RELBLK and BLKDSP fields of the workspace and the Bitmap Handler executes an RTS to return control to the calling routine. If the end of the directory is reached without finding a free directory, a "-1" is moved to the RELELK field of the workspace to signal this fact and an RTS is done.

When file space allocation is requested, the number of bytes to be allocated is placed in the NBRREQ field. The Bitmap Handler calls subroutine VALCNK, which converts this number to the number of chunks that must be allocated (i.e. $\text{ceiling}[\text{bytes}/64]$). The Bitmap Handler then looks through the bitmap until it finds enough

contiguous free chunks to satisfy the request. These bits are then cleared, the address of the lowest numbered block allocated is placed in RELBLK and the address of the lowest byte allocated in this block is placed in the BLKDSP field. The higher level routines therefore receive the starting address of a contiguous segment of disk space guaranteed to be at least as large as they requested, but possibly spread across several blocks.

File space deallocations are done in a similar fashion. The number of bytes to be deleted is placed in NBRREQ, but with bit 15 set to "1" as a flag that a delete is being requested. The starting address of the disk string to be deleted is placed in the RELBLK and BLKDSP fields, as with directory deletes. The Bitmap Handler then translates the number of bytes into the corresponding number of chunks and sets the proper bits in the bitmap back to "1" to show that space free.

Before attempting to understand the address translation mechanism, it is perhaps wise to reconsider the bitmap layout. There are 16 bits/word in the bitmap, 8 chunks/block of disk. Therefore each word in the bitmap covers two blocks of disk chunk space. Since blocks are allocated from the highest block down and we associate the first word in the bitmap with the last block of the file, the difference in bytes between any byte and the first byte of

the bitmap is the number of blocks down from the highest block in the disk file. The low-order bit in a byte corresponds to the chunk starting at displacement 0 in that block, the next one to the chunk starting at offset 64 (bytes),...,the high-order bit (bit 7) corresponds to the chunk starting at byte 448 within the block. In order to convert a bitmap address into a disk address, therefore, we must subtract the offset in bytes into the bitmap from the highest block number in the file to get the block number. To get the displacement in block, we need to multiply the relative bit position within the byte by 64.

3.6 SEARCH SUPERVISOR

The Search Supervisor (SRCHSP) module is the scheduling and control module that orchestrates the performance of the Merge Routine, Set Handler, Index and Postings Handler, and Full-Text Searcher modules in the execution of a search.

The Search Supervisor may be performed by either the FIND Statement Parser, the PRINT Statement Parser, or the MAKE Statement Parser and is passed the address of a Search Supervisor Table (Fig. 3.6.1.1) in register R0. This table contains all the pertinent information collected from the user's query by the command parser.

3.6.1 SEARCH SUPERVISOR TABLE

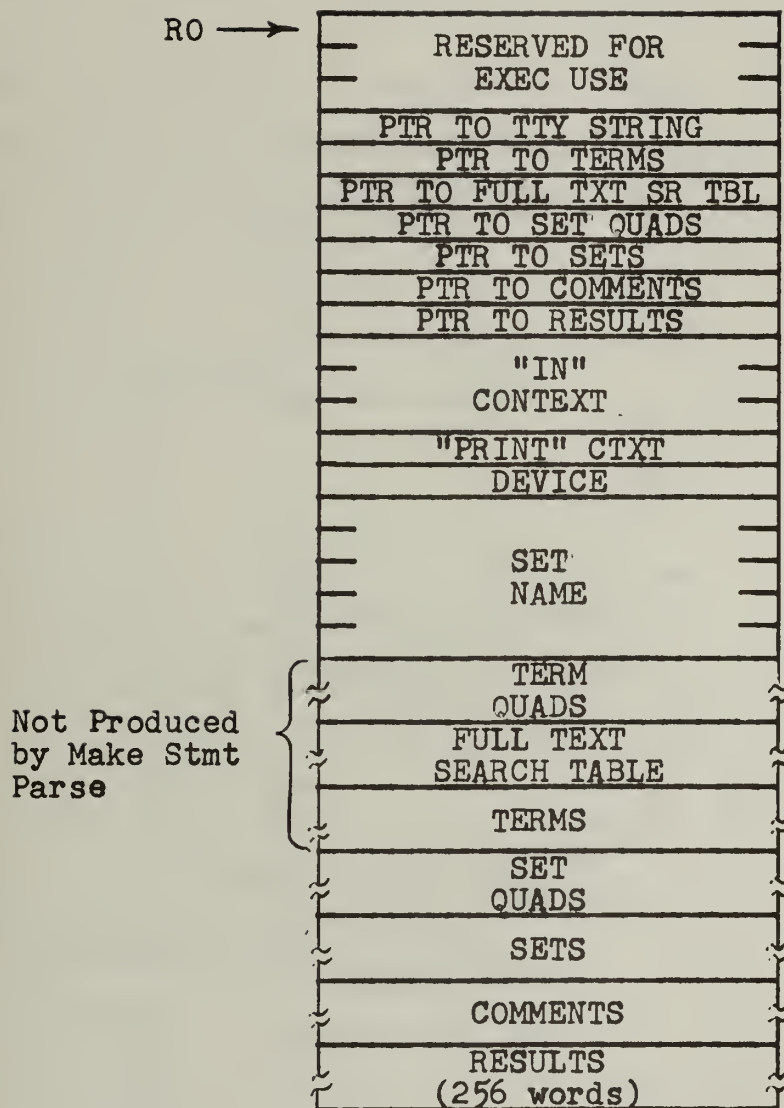
Since the primary function of the Search Supervisor (and the rest of EUREKA, for that matter) is to perform the operations described in the Search Supervisor Table, we shall take a detailed look at its contents and form.

Referring to Fig. 3.6.1.1, we see that the first three words pointed at by R0 are reserved for EXECUTIVE use. This actually reflects an earlier incarnation of the EXECUTIVE, and these words are not currently used.

The next word in the table, "PTR TO TTY STRING", is the address of the buffer containing the text of the user's query. The next six words are all pointers (memory addresses) to various information blocks within the table that shall be described later.

The next block, "'IN' CONTEXT", is a three word descriptor identifying the contexts in which the term expression must occur. Similarly, the block labeled "'PRINT' CONTEXT" is a one word descriptor identifying the contexts to be printed from documents that satisfy the search request.

The word labeled "DEVICE" is a descriptor specifying whether the information is to be printed on the line printer or displayed upon the user's screen.



SEARCH SUPERVISOR TABLE

Fig. 3.6.1.1

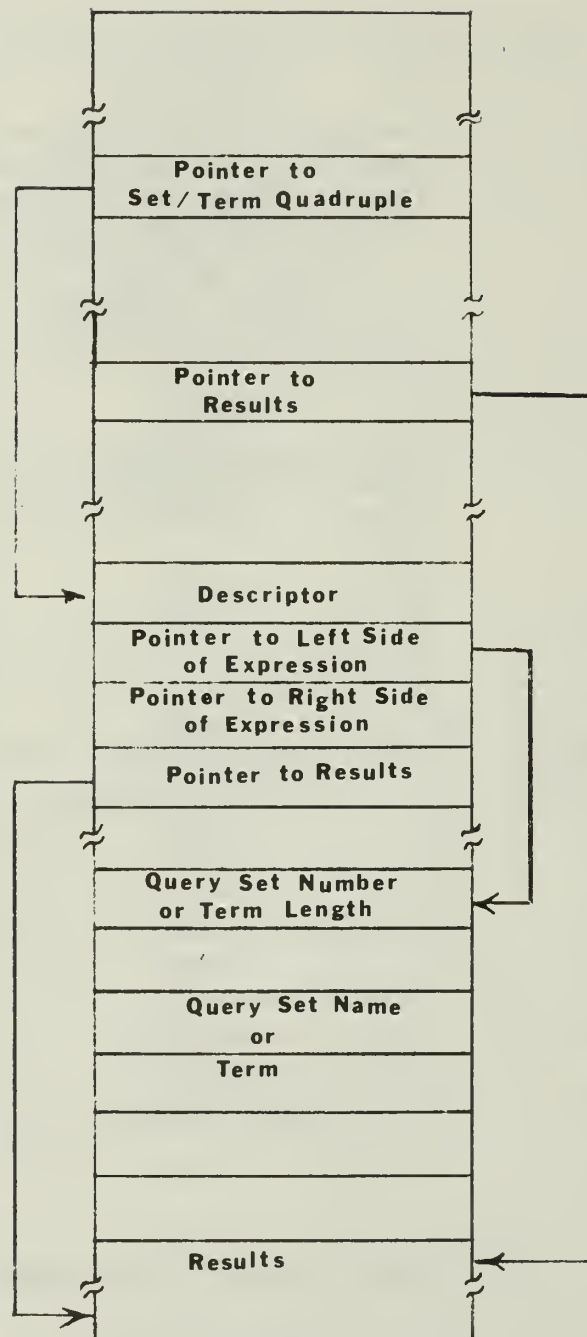


Figure 3.6.1.2 Term/Set Quad Detail

The five word block "SET NAME" is the mnemonic name the user wishes to have attached to the resulting query set.

"TERMS" and "TERM QUADS" are the two blocks that specify the search terms for this query and in what relationship they are to occur. Fig. 3.6.1.2 describes the structure of these two blocks. The "TERM QUADS" block holds a set of four word descriptors which form a binary operation tree that describes the search expression. The first descriptor in the block is the root node of the operation tree.

Each descriptor is made up of four words, the first being a word of bit-flags, the second and third being pointers to the left and right hand terms of the expression (with respect to the operator described at this node), and the fourth being a pointer to the address at which the results of this operation are to be placed. The bit-flag word is broken down as follows:

Bit 15,14: Operation to be performed;

00 => OR

10 => AND

11 => AND NOT

Bit 11 : 1 => Suffixing to be performed on left hand side term.

Bit 10 : 1 => Prefixing to be performed on left hand side term.

Bit 8 : 1 => Left hand side pointer points to another node (term

quad) rather than a leaf (term).

Bit 3 : same as bit 11, only for right hand side.

Bit 2 : same as bit 10, only for right hand side.

Bit 0 : same as bit 8, only for right hand side.

Note that bits 8, 10, and 11 are bits 3, 2, and 0 of the high-order byte.

The node-leaf selector bit allows us to handle cases of the form:

FIND 'A' * 'B' * 'C'

Since we use binary operations exclusively, we must first AND together the list of documents responding to the first search term with the list of documents responding to the second search term, thus producing a temporary result to be ANDed with the list of documents responding to the third search term. Whenever the node-leaf bit is set to 1 for either the left or right hand side the "TERM POINTER" points to the set quad of the operation that must be performed in order to generate the temporary result list needed to perform the operation described in the current node. If the right/left node-leaf bit is set to 0 then the right/left term pointer is the address of the search term to be used in the right/left hand side of the current operation. The "TERMS" block contains all the terms pointed at by the term pointers just described. The terms are laid out sequentially in the "TERMS"

block, each consisting of a length word followed by the text of the term. The term pointers actually point to the length words.

The result pointer in each term quad is filled in by the Search Supervisor upon completion of the operation specified in that node and is used whenever a parent node term pointer points at the current node.

"SET QUADS" and "SETS", which describe the set expression for this search, are structured in the same form as "TERM QUADS" and "TERMS". The only significant differences are that the prefixing-suffixing bits in the bit-flag words are meaningless here and the "SET POINTER" points to a six word block containing either a query set number in the first word thereof or a query set name in the last five words. The indirect (node-leaf) bit, results pointer, and op code bits are the same as for the "TERM QUADS".

3.6.2 SEARCH SUPERVISOR OPERATION

Now that we have an understanding of the Search Supervisor Table the description of the Search Supervisor is relatively trivial. The only difficulties occur in attempting to describe the handling of the various "special cases" that can occur. We shall therefore first look at the main structure of the code and then go back and describe how the "special case" handlers fit within the

framework of the body of the module.

The first action of the Search Supervisor (aside from some housekeeping) is to start up the Set Expression Evaluator (STEV), which evaluates the set expression as contained in the "SET QUADS" and "SETS" blocks of the Search Supervisor Table. While the Set Expression Evaluator is in progress the Search Supervisor does some initialization of working lists for use in evaluating the term expression. As soon as the Set Expression Evaluator finishes, the Search Supervisor evaluates the term expression by use of a loop which effectively traverses the "TERM QUADS" tree in postorder, using the Index and Postings Handler (IPHNDL) to read in the postings list for each leaf (search term), and the Merge Routine (MERGE) to perform the operation specified at each node of the "TERM QUADS" tree.

The Search Supervisor then performs the Full-Text Searcher (FTSRCH) which determines whether any of the documents require full-text searching and, if so, performs the search. Refer to Sec. 3.10 for details of the operation of the Full-Text Searcher.

Upon completion of the Full-Text Searcher execution the Search Supervisor prints the message:

n DOCUMENTS POSTED TO THIS SET

and then constructs a Set Handler Table describing the results of

this search and performs the Set Handler in order to save a record of this search for future use by the user.

All that remains then is the not inconsiderable task of cleaning up by freeing all the temporary lists used in the term expression evaluation and freeing all the disk space used by the Merger. A \$RETN trap is then executed to return control to the parser routine that initiated (via a \$PRFM trap) the Search Supervisor.

Now we shall look at the special cases. If the Search Supervisor has been performed by a PRINT Statement then no term expression exists and the section of code that evaluates this expression must be skipped. The Search Supervisor must, however, handle print requests that access user comments. This requires a call to the Set Handler to get the list of all documents which have user comments attached to them. The Search Supervisor must also skip over the call to the Set Handler that saves the search results in order to avoid generating a new query set from a PRINT statement.

MAKE Statements present a similar problem in that they have no term expression to be evaluated, but this is easily handled by skipping the term expression evaluation process.

The last major "special case" to consider is the case where one or more search terms contain no alphanumeric characters and therefore do not have entries in the index file. If this case arises, the Search Supervisor must construct a set list consisting of all the document accession numbers in the entire database for this term and force full-text searching of all of them.

For further details, refer to the program listing.

3.7 SET EXPRESSION EVALUATOR

The Set Expression Evaluator (STEV) performs the function of evaluating the set expression contained in the "SET QUADS" and "SETS" blocks of the Search Supervisor Table (see Fig. 3.6.1.1; also refer to the preceeding section of this report for a description of the table layout). The address of the Search Supervisor Table is passed to this routine in register R0.

Once one understands the structure of the "SETS" and "SET QUADS" sections of the Search Supervisor Table the operation of the Set Expression Evaluator is self-evident. It traverses the operation tree in postorder, using the Set Handler to retrieve document lists (query set lists) from the user's file for the leaves and the Merger (MERGE) to perform the operations specified at the nodes. The final result is put in the area of the Search Supervisor

Table labeled "RESULTS".

3.8 INDEX AND POSTINGS HANDLER

The purpose of the Index and Postings Handler (IPHNDL) module is to determine in which documents the user's search terms occur so that we need only consider those documents rather than searching the entire database to find documents that satisfy the user's search expression.

3.8.1 FILES MANIPULATED BY THE INDEX AND POSTINGS HANDLER

Before attempting to fathom the details of the Index and Postings Handler, let us consider the file structures manipulated by it. This file structure consists of two hash tables, HASH1 and HASH2 (read in from files HASH1.XXX and HASH2.XXX by IRINIT); the index file, INDEX.XXX; and the postings file, PSTNG.XXX. The "XXX" file extension on the file name is used to distinguish between various versions of EUREKA and also between various databases.

The two hash tables are used to get a disk address in the index file. HASH1 is used to hash on the first letter of a token and HASH2 is used to hash on the second letter of the token. The sum of the values obtained (via a process explained in Sec. 3.8.2) from HASH1 and HASH2 is used to give a disk address in the index file

(INDEX) where terms beginning with these two characters are indexed. This section of the index file is then searched linearly for a match to the entire token until the index file entry is lexicographically less than the token. If a match is found a pointer into the postings file (PSTNG) is obtained from the index file. The postings file contains the list of all documents in which the token under consideration occurs. Each entry in the postings file contains a document accession number, context bits to describe the context(s) in which the token occurs, and a count of the number of occurrences of the term within the document.

Let us now look at the layouts of the files and tables in a semi-tabular format.

3.8.1.1 HASH1 TABLE

Table name: HASH1

Size: 256 words

Content: Each word contains full word value to hash the character which indexes it. If the value of the word is FFFF base 16, then the character does not exist in the index.

3.8.1.2 HASH2 TABLE

Table Name: HASH2

Size: 256 Bytes

Content: Each byte contains a byte value to hash the character which indexes it. If the value of the byte is FF base 16, then the character does not exist in the index.

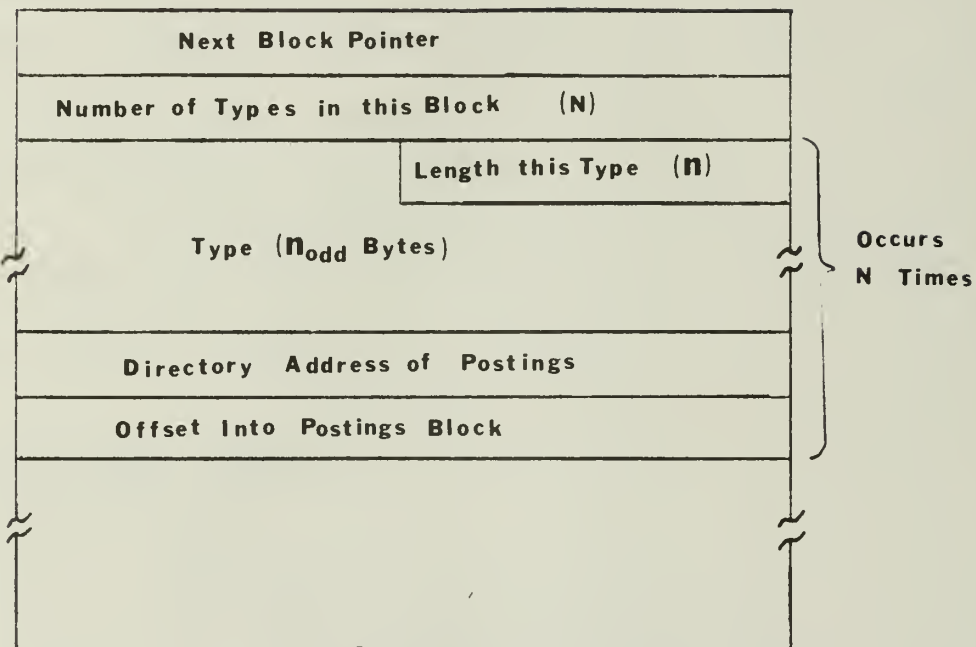
3.8.1.3 INDEX FILE

File Name: INDEX.XXX

Type: Contiguous

Blocking Factor: 2

Format:

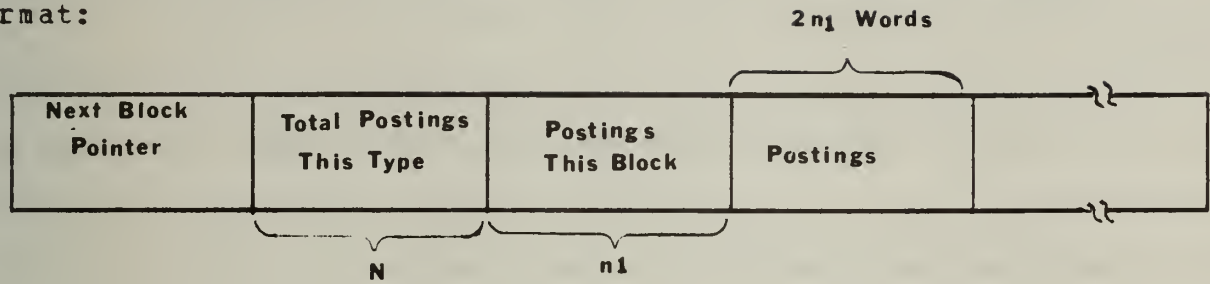


3.8.1.4 POSTINGS FILE

File Name: PSTNG.XXX

Type: Contiguous

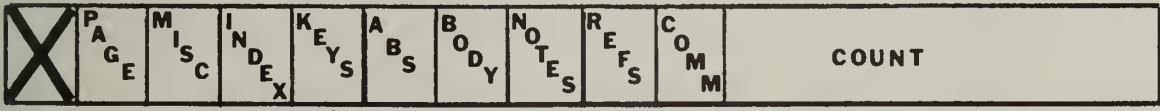
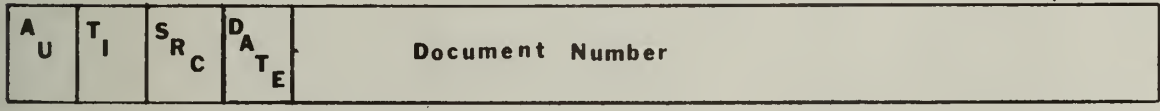
Format:



N > n1 implies postings for a type are split across blocks. In this case, the next block has the following format:



Each posting consists of two words in the following format:



3.8.2 OPERATION OF THE INDEX AND POSTINGS HANDLER

Now that we have analyzed the files manipulated by the Index and Postings Handler, let us look at the operation of the routine itself.

Upon entry, register R0 should point at a table of six words containing:

- 1) A prefix/suffix descriptor for the term.
- 2) A pointer to the term.
- 3) A pointer to where the postings are to be placed.
- 4) and 5) A two-word context descriptor.
- 6) A pointer to the Logon Block.

The prefix/suffix descriptor contains only two bits of useful information, if bit 2 is on (i.e. equals 1), then prefixing is to be used; if bit 3 is on, then suffixing is to be used. If both bits are on, then both prefixing and suffixing are to be used. The pointer to the term points to a term in the "TERMS" block in the following format: One word containing the length in characters (bytes) of the term, followed immediately by the text of the term. The context descriptor is in the standard form shown in Sec. 3.8.1.4.

After the usual housekeeping the Index and Postings Handler first checks to see if the term contains any non-alphanumeric characters. If it does, then full text searching will be required of all documents containing as a substring any token within the term. If, as is normally the case, there are no special characters in the term, the Index and Postings Handler checks to see if prefixing or suffixing has been specified for this term in the descriptor word. If either or both have been specified then control is passed to one of three special purpose routines (PREFIX, SUFFIX, and BOTH) which shall be described later. In the simplest case (where the user has requested a term with no prefixing, suffixing, or special characters) the Index and Postings Handler hashes on the first two characters of the term to obtain the address in the Index File to begin searching for an exact match to the search term. The hash is done by treating the bytes containing the characters being hashed as if they were numeric values. The first character is multiplied by two (i.e. shifted left one bit) and is used as an index into table HASH1. A one word value is retrieved from the indexed location in HASH1. If the value is FFFF base 16, then the character doesn't exist in the index. The second character is then treated similarly, retrieving a byte value from HASH2 which is added to the word value retrieved from HASH1 (unless it is FF base 16, the flag that a character is non-existent) to obtain a disk address in

the index file at which terms beginning with this bigram are located. If either value has been flagged as non-existent the Index and Postings handler marks this fact and immediately executes a \$RETN trap. If a valid disk address has been obtained by the hash, the Index and Postings Handler uses the subroutine GETNDX to retrieve the index listing for the term in question. Once the index entry has been found (if it exists) the Index and Postings Handler uses the subroutine GETPST to retrieve the postings (list of document accession numbers) for this term and calls the Merger to merge this list with any previously generated lists (this occurs primarily when handling special cases such as prefixing).

Once the final postings list has been constructed it is read into the buffer specified in the six word descriptor table. If the results list is too long to fit in the buffer, then only the first block is read in, with a link pointer to the remainder on disk. After some fairly messy housekeeping a \$RETN trap is done.

The special subroutines SUFFIX, PREFIX, and BOTH handle finding all truncated matches for terms and merging the postings lists of each newly found posting into the list of ones already found. These routines use the common exit routine to clean up after execution and do the \$RETN trap that returns control to the Search Supervisor.

3.9 MERGER

The Merger (MERGE) is used to perform Boolean operations (AND, OR, and AND NOT) on lists of document accession numbers. A subsidiary function is the allocation/deallocation of scratch disk space for itself and for the Search Supervisor, Index and Postings Handler, and the Full-Text Searcher. Parameters are passed to this module in the form of a nine word long parameter list containing:

Word 0.....operation descriptor
 Word 1.....pointer to left hand side list
 Word 2.....pointer to right hand side list
 Word 3.....pointer to results buffer
 Words 4&5.....two-word long context descriptor
 Word 6.....pointer to left hand side buffer
 Word 7.....pointer to right hand side buffer
 Word 8.....disk address of result list overflow.

In word 0, the operation descriptor, only the high-order byte is meaningful. The bits of this byte have the following meanings:

Bits 15,14 : Binary operation code as described in Section 3.6.1.

Bit 11 : 1 => Free the scratch disk space whose starting address is located in word 8 of the parameter list (bytes 15,16).

Bit 10 : 1 => Allocate a scratch disk space and place the starting address in word 8 of the parameter list.

Words 1 and 2, the left and right hand side list pointers, are the starting memory addresses of the two document accession number lists to be merged. The first word of each list contains the number of document accession numbers (postings) in the list, while the second word contains the number of postings in this block. These words are followed by the document accession list in the standard two-word long descriptor format described in Sec. 3.8.1.4.

Word 3 points to the memory buffer in which to store the result list. This buffer is only one block long, so if the result list is over one block long only the first block of the list is stored here. The rest of the list is stored on disk as a one-way linked list with the starting block number stored in word 8 of the parameter list.

Words 4 and 5 are a standard context descriptor that is used for setting context bits in entries in the result list for use by the Full-Text Searcher.

Words 6 and 7 are pointers to the head of the buffer used by the Index and Postings Handler for reading in the information from the Postings file. Words 1 and 2 are addresses within this buffer. The buffer addresses are provided in case the posting spreads across more than one block.

Word 8 is used to return disk addresses of result list overflow lists, the address of freshly allocated disk space, and to receive the address of scratch disk buffers to be deallocated.

The first action of the Merger is the decoding of the operation decriptor. If the request in the parameter list is for disk buffer allocation/deallocation then subroutine BMAP is called. BMAP is a straightforward bitmap handler which keeps track of which disk buffers are in use. As soon as BMAP has updated its bitmap and placed its result in the parameter list a \$RETN trap is executed to return control to the calling routine.

Boolean operations on document accession number lists are performed in seperate loops (one for each operation) that compare the two lists on an element-by-element basis, generating the result list with correct context bits set as it does so. After the two lists have been merged into a result list a \$RETN trap is executed, returning control to the calling routine.

It is hoped that this routine will be replaced by a hardware merger at some time in the near future.

3.10 FULL-TEXT SEARCHER

The Full-Text Searcher (FTSRCH) module does all the full-text searching, browsing, and text display for the user. This module is made up of three main routines: FTSRCH, the full-text searching routine; BROWSE, the browse mode handler; and PRNTR, the document text display routine. It is initiated (via a \$PRPM trap) by the Search Supervisor and receives the address of the Search Supervisor Table in register R0.

3.10.1 FULL-TEXT SEARCHING ROUTINE The Full-Text Searching routine (FTSRCH) is called during each search after the Index and Postings Handler has constructed a list of all documents that contain the proper Boolean conjunction of search terms. The Full-Text Searcher sorts this list of documents into highest-count-field-first sequence and rewrites it to disk. If the user has requested that all text printed for this query be displayed upon the line printer a \$LOCK trap is executed to lock the line printer. The Full-Text Searcher then determines what type of search (if any) is to be performed on documents. Next the Full-Text Searcher enters a loop that reads in the directory for each document in the list, one at a time. Tests are then made to determine if any type of full-text search or text print is to be performed on this document. If a full-text search is to be performed, control is passed to the appropriate controlling

loop for the type of search to be performed. If no search is required, a JSR is made to the Text Print (PRNTR) routine. The search controlling loops set up parameters for the subroutine LEVEL1, which does the actual searching. Each time a "hit" is found in the text being searched, LEVEL1 returns control to the parent loop, which handles coordinating Boolean conjunctions of terms within contexts, etc. Whenever some text that satisfies the search request is found, control is passed to the Text Printer routine, which formats and displays the text if a print has been specified.

3.10.2 TEXT PRINTER ROUTINE

The Text Printer routine (PRNT) does all text display for the EUREKA system. It is called by the Full-Text Searcher routine and the Browse Mode Handler. If the user has requested that some term(s) be found in the same paragraph and that the sentence in which they occur be printed; or that they be found in the same sentence and that the paragraph in which they occur be printed, then this routine is passed the starting and stopping addresses of the clause that satisfies the Print Clause. Under any other combination of requests, the Text Printer gets all needed information from the Search Supervisor Table.

The Text Printer has several special subroutines used for handling the:

"FIND <Term Expression> IN SENTENCE PRINT PARAGRAPH"

type of situations. These routines utilize "inside knowledge" about starting and stopping addresses in the text, etc. to set up parameters for the regular formatting and marking routines and then perform them just as the main body of the Text Printer does.

The main body of the Text Printer first goes through a series of tests to see if individual contexts are to be printed. On each "hit", the appropriate context is moved into the parameter areas of the workspace and the PRINT1 subroutine, which handles the mechanics of printing one context, is called. For some contexts, such as a SENTENCE print during an "IN PARAGRAPH" find, the special-purpose routines described before must be called.

The PRINT1 subroutine uses the subroutine RDTXT to read in the text containing the context to be printed, the subroutine MARK to mark the text to be displayed, and the subroutine FORMAT to actually format and display the marked text. Mark does little more than stick a special fern in front of search terms to mark them for FORMAT and will not be discussed in any greater detail. FORMAT handles the mechanics of moving the text to be displayed into the print buffer, deleting ferns, moving asterisks to column one of any

buffer that contains a mark fern, and actually displaying the text. After the text has been displayed, a JSR is made to the Browse Mode Handler (BROWSE), unless the information is being displayed upon the line printer, in which case the JSR is skipped and an immediate RTS is done.

3.10.3 BROWSE MODE HANDLER

The Browse Mode Handler (BROWSE) controls all interaction with the user while text display is in progress. It prompts the user each time a context is printed and examines his reply to see if any browse commands have been entered. If no browse mode commands have been entered an immediate RTS is done to return control to the Text Printer routine. If the user has entered a command that requests printing of another context or previous/succeeding sentence or paragraph, the Browse Mode Handler must take care of the mechanics of retrieving the text to be displayed, setting up parameters for the subroutine FORMAT, and calling it to have the text printed. If the user has entered a comment string to be attached to the document currently being viewed the Browse Mode Handler must build a Set Handler Table containing the Logon Block pointer, document accession number (with bit 15 set to 1 to flag it as a document), and pointer to the comment string. The Browse Mode Handler then performs the Set Handler and then re-prompts the user for another command.

3.11 SET INFORMATION PRINTER

The Set Information Printer (INFOPT) module is used to retrieve information from the user's personal file and display it. Its function, therefore, is primarily that of calling the Set Handler to retrieve information from the user's file, format it, and display it upon either the user's terminal or upon the line printer.

The first operation of the Set Information Printer is some housekeeping which includes workspace allocation, line buffer header initialization [1], locking (via a \$LOCK trap) the line printer if necessary, and various buffer initialization. The Set Information Printer then determines whether the information to be printed is a macro or a query set and proceeds to the proper section of the module. We shall first look at the case of a user macro print.

In the case of a user macro print the Set Information Printer first checks to see if the macro identifier in the Set Handler Table is "ALL". If it is, the user has requested that all macro definitions in the user's file be displayed. In this case the user must first request a list of all the macro identifiers in the user's directory from the Set Handler. Once it has this list it goes into a loop which moves one macro identifier at a time into the Set Handler Table and JSR's into the display subroutine once for each macro. If the identifier is not "ALL", the user has requested to

see a single macro definition and the Set Information Printer needs only perform (via a JSR) the display subroutine once and then exit by executing a \$RETN trap.

The query set print section of the module works in much the same fashion as the macro print. All query set print requests are assumed to be of the form:

```
PRINT <Query ID 1> TO <Query ID 2>
```

This form is reflected in the use of a modified Set Handler Table for passing parameters to the Set Information Printer module. This table looks like a Type I Set Handler Table with a second Query Set Descriptor (describing <Query ID 2>) following the first. If the user has requested the display of a single query, the second Query Set Descriptor is zeroed. If the user has specified either of the <Query Set ID>'s via a mnemonic name the Query Set Information Printer requests the query number of that query set from the Set Handler in order to use it as a bound on the printing loop. Once the Query Set Information Printer has both query numbers it clears the set name field of the Set Handler Table, moves the lower of the two query numbers to the query number field of the table, and subtracts one from the query number field. It then enters a loop that:

- 1) Adds one to the query number field of the Set Handler Table;
- 2) Compares the query number to the upper bound of the print

request: if it is less than or equal to the upper bound it performs the display subroutine once and loops back to (1); if it is greater than the upper bound the Set Information Printer jumps to the exit routine.

The display subroutine (PRINT) is referenced by both the macro print section and the query set print section. It receives as input a Type I Set Handler Table containing the correct identifiers/buffer pointers to read in the information to be printed. The display routine calls the Set Handler once for each pertinent block of information to be retrieved (query/macro text, query set list, comments). When the Set Handler returns control to the display routine it formats the data for display, prints it wherever the user has requested that it be printed, and executes an RTS instruction to return control to the controlling loop. If the Set Handler has returned an error message of "INVALID SET ID" on the stack, the display routine merely clears the stack and does an RTS, returning control to either the macro print loop or the query set print loop, thus discarding the error message. All other error messages are propagated back up the tree of tasks. This allows the Query Set Information Printer to handle cases where the user requests to see all query sets between <Query Number L> and <Query Number N> where some <Query number M>, $L < M < N$ has been deleted.

The exit routine of the Information Print routine checks to see if the line printer has been locked and, if so, unlocks it by doing a \$UNLK trap. Following this, a \$RETN trap is executed to return control to the PRINT Statement Parser.

APPENDIX A

Descriptions of Context Terms

The following are the current definitions of the context terms:

SENTENCE.....any text between two periods (.).

PARAGRAPH.....any text appearing between two paragraph ferns.

COMMENTS.....user written and assigned comment strings.

REFERENCES.....any references listed by the author of the paper.

NOTES.....not used at present.

TEXT.....abstract, body, notes, and references.

KEYS.....not used at present.

INDEX.....not used at present.

MISC.....junk.

PAGES.....journal page numbers on which article occurred.

DATE.....date written for dated papers.

SOURCE.....journal from which taken.

TITLE.....title of document.

AUTHOR.....author of document.

DATA.....includes author, title, source, date, pages, and misc.

ARTICLE.....everything.
DOCUMENT.....same as article.
BODY.....text of article excluding abstract.
ABSTRACT.....abstract of article.

Obviously not all of the contexts are applicable in all cases, and in fact are presently somewhat sparsely filled in.

APPENDIX B
Error Messages

INVALID COMMAND:

EUREKA didn't recognize the first word of your command.
check for spelling or abbreviation errors.

SET NAME > 10 CHARACTERS:

You have attempted to attach a name with 11 or more
letters to a query set. This is not permitted. Watch
for concatenated words on multiple line commands.

ILLEGAL OR IMMORAL USE OF QUOTES:

Check for unmatched single quotes, i.e. ' . Also check
for single quotes used in improper places.

INVALID WORD FOLLOWS "ALL":

EUREKA cannot parse the part of your command that comes
after the word "all".

CANNOT DELETE DOCUMENT:

EUREKA will not allow you to delete a document, only the
comments attached to the document.

ILLEGAL USE OF BRACKETS:

EUREKA has discovered some brackets where it doesn't
expect them.

QUERY OR DOCUMENT NO. TOO BIG:

You have just input a query/document number that is far bigger than the number assigned to any query or document in the EUREKA system.

MISSING SET NAME IN CHANGE STMT.:

EUREKA cannot find the name of the query set you wish to have changed.

MISSING KEYWORD OR SETNAME IN CHANGE STATEMENT:

EUREKA cannot find "TO" in your CHANGE statement. Check to see if you have two Set Names separated by " TO ".

TOO MANY SET NAMES:

EUREKA is confused. There are too many names there.

QUERY OR DOCUMENT NO. IS NOT NUMERIC:

EUREKA has found a string of letters in a place it expected only numbers. Be sure you haven't typed an "O" instead of an "0".

NO SET NAME IN DELETE STATEMENT:

Did you tell EUREKA what you wanted deleted? It doesn't think you did!

ILLEGAL USE OF BRACKETS:

Check the usage of brackets ("[" or "]") for validity.

NAME OF DOCUMENT CANNOT BE CHANGED:

You may not assign a name to a document. A close approximation is to use the make statement to create a query set with only the desired document in the set.

INVALID CHARACTERS IN SET NAME:

A Set Name cannot contain any symbols other than letters of the alphabet or numbers; the first letter of the Set Name must be non-numeric.

MISSING SET NAME:

You've left out the name of a Set somewhere.

LOGICAL END OF STATEMENT REACHED PREMATURELY:

EUREKA thinks you started something that you didn't finish. Check your brackets, etc.

PARSER TABLE OVERFLOW:

Program couldn't handle your expression - please use smaller queries to achieve your final result.

ILLEGAL SUCESSOR:

You probably have either left out an operator (*, +, or -), or have two of them with nothing in between them.

ILLEGAL CONTEXT:

EUREKA doesn't recognize the context you want it to use. Check your spelling or abbreviation.

ILLEGAL SET NAME:

Check to see if you have used a reserved word or invalid characters.

TOO MANY LEVELS OF PARENTESSES:

Your expression is too complicated. Break it up into several FIND/MAKE statements.

TOO MANY TERMS OR PARENTHESES:

Same as previous message.

ILLEGAL TERM:

Most likely explanation is that you have too many or too few single quotes (') in your search expression.

TOO MANY DISJUNCTIONS:

Expression too complicated. Use several smaller queries.

ILLEGAL SET OR DOCUMENT NO.:

The number is either non-numeric or too big.

EXPRESSION TOO COMPLICATED:

Use several smaller queries and/or MAKE statements to accomplish your grand design.

INVALID USER ID:

Check your spelling. If you cannot log on, see a systems programmer.

INVALID BITMAP:

Some of your records may be missing. Notify a systems programmer.

INVALID BOOLEAN CONJUNCTION OF SETNAME-#'S:

You've probably left out or put in an extra Boolean operator (*,+, -).

YOU HAVE ATTEMPTED TO RENAME A NON-EXISTENT SET:

Self explanatory. Check your spelling.

YOU HAVE ATTEMPTED TO READ A NON-EXISTENT SET:

Same as previous error message.

YOU HAVE ATTEMPTED TO COMMENT A NON-EXISTENT SET:

Same as previous error message.

YOU HAVE RUN OUT OF DISK SPACE. PLEASE DELETE SOMETHING:

Your disk space is completely full - EUREKA cannot find enough space to finish processing your last command. Delete some sets and/or comments before proceeding.

SET EXPRESSION NOT VALID:

You have probably left out a Boolean operator (*,+, -) in your set expression.

NO "LAST" SET EXISTS:

You have attempted to access the "LAST" set, which has been deleted or somehow changed.

ILLEGAL MACRO NAME:

Macro names must obey the same regulations as query set names.

YOU MUST LOGON!:

No queries may be entered until you have logged on.

SYSTEM ERROR:

Program error - call system programmer immediately.

REFERENCES

1. Digital Equipment Corporation, "Disk Operating System Monitor Programming Handbook", 1972.
2. Milner, J.M., "A Multiprocess, Multiuser Executive for an Experimental Information Retrieval System", M.S. thesis, University of Illinois Department of Computer Science Report Number 75-736, August 1975.

BIBLIOGRAPHIC DATA SHEET		1. Report No. UIUCDCS-R-76-779	2.	3. Recipient's Accession No.
4. Title and Subtitle DESCRIPTION OF AN EXPERIMENTAL ON-LINE, MINICOMPUTER-BASED INFORMATION RETRIEVAL SYSTEM			5. Report Date February 1976	
7. Author(s) John Keith Morgan			8. Performing Organization Rept. No. UIUCDCS-R-76-779	
9. Performing Organization Name and Address University of Illinois at Urbana-Champaign Department of Computer Science Urbana, Illinois 61801			10. Project/Task/Work Unit No.	
			11. Contract/Grant No. US NSF DCR73-07980 A02	
2. Sponsoring Organization Name and Address National Science Foundation Washington, D. C.			13. Type of Report & Period Covered Master's Thesis - 1976	
			14.	
5. Supplementary Notes				
6. Abstracts Traditionally, information retrieval systems have provided little more than titles or abstracts of documents in response to user queries. This thesis describes an experimental information retrieval system that provides a framework for research in providing users access to the entire text of documents.				
7. Key Words and Document Analysis. 17a. Descriptors Database systems Documentation Information retrieval Inverted files Nonnumeric processing Query languages User aids				
8. Identifiers/Open-Ended Terms				
9. COSATI Field/Group				
18. Availability Statement PLEASE UNLIMITED			19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 135
			20. Security Class (This Page) UNCLASSIFIED	22. Price

APR 6 1978



UNIVERSITY OF ILLINOIS-URBANA
510.84 IL6R no. C002 no.778-781(1975
Theoretical limitations on the use of pe



3 0112 088402505